(12) UK Patent Application (19) GB (11) 2 339 040 (13) A

(43) Date of A Publication 12.01.2000

(21) Application No 9907221.7

(22) Date of Filing 29.03.1999

(30) Priority Data
(31) 09053127 (32) 31.03.1998 (33) US

(51) INT CL[7]
G06F 9/302

(52) UK CL (Edition R)
G4A AVL

(56) Documents Cited
WO 97/22924 A1 WO 97/22923 A1 WO 97/22921 A1

(58) Field of Search
UK CL (Edition Q) G4A AVL
INT CL[6] G06F 9/302

(71) Applicant(s)
Intel Corporation
(Incorporated in USA - Delaware)
2200 Mission College Boulevard, Santa Clara,
California 95052, United States of America

(72) Inventor(s)
Patrice Roussel
Ticky Thakkar

(74) Agent and/or Address for Service
Langner Parry
High Holborn House, 52-54 High Holborn, LONDON,
WC1V 6RR, United Kingdom

(54) Abstract Title
Executing partial-width packed data instructions

(57) In a data processing system arranged for executing scalar packed data instructions, a processor includes a plurality of registers, a register renaming unit coupled to the plurality of registers, a decoder coupled to the register renaming unit, and a partial-width execution unit coupled to the decoder. The register renaming unit provides an architectural register file to store packed data operands each of which include a plurality of data elements. The decoder is configured to decode a first and second set of instructions that each specify one or more registers in the architectural register file. Each of the instructions in the first set of instructions specify operations to be performed on all of the data elements stored in the one or more specified registers. In contrast, each of the instructions in the second set of instructions specify operations to be performed on only a subset of the data element stored in the one or more specified registers. The partial-width execution unit is configured to execute operations specified by either of the first or the second set of instructions.

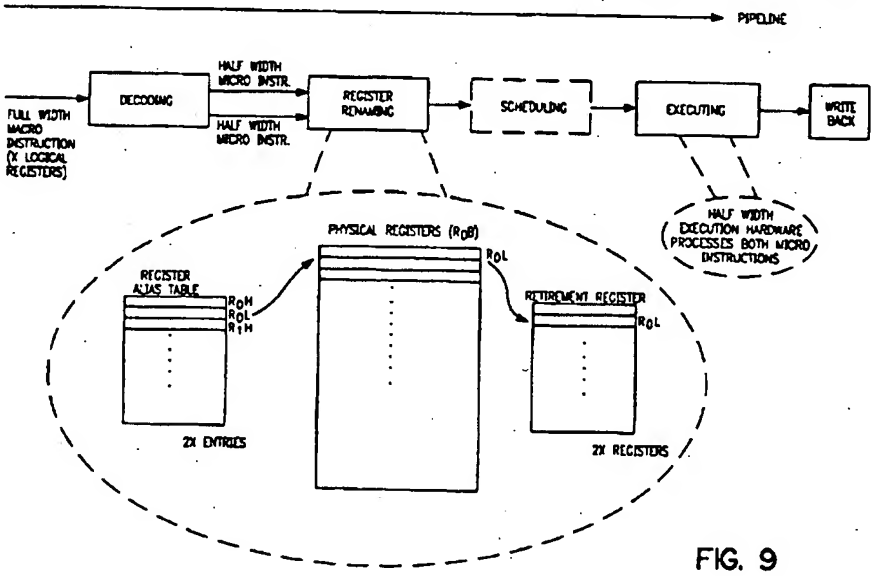


FIG. 9

GB 2 339 040 A

At least one drawing originally filed was informal and the print reproduced here is taken from a later filed formal copy.

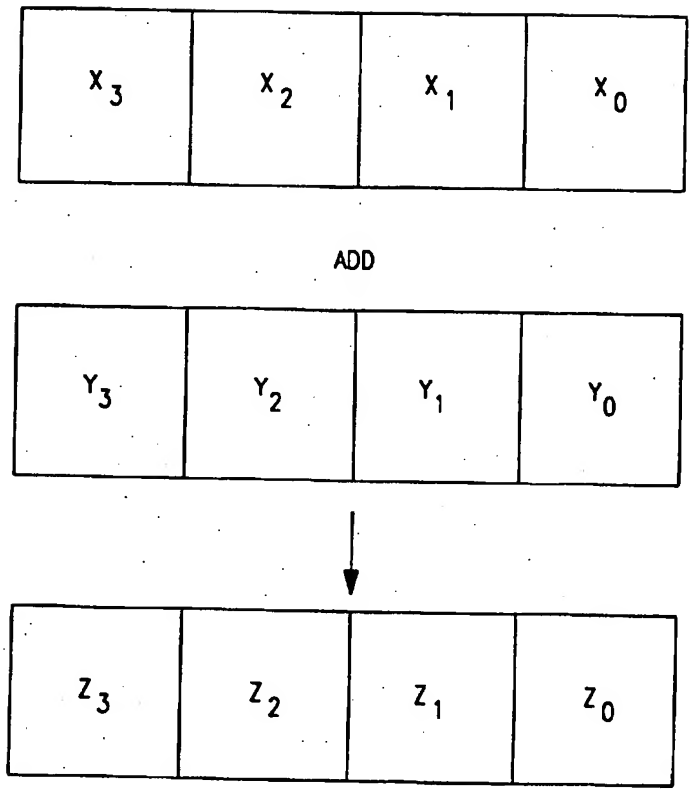


FIG. 1
PRIOR ART

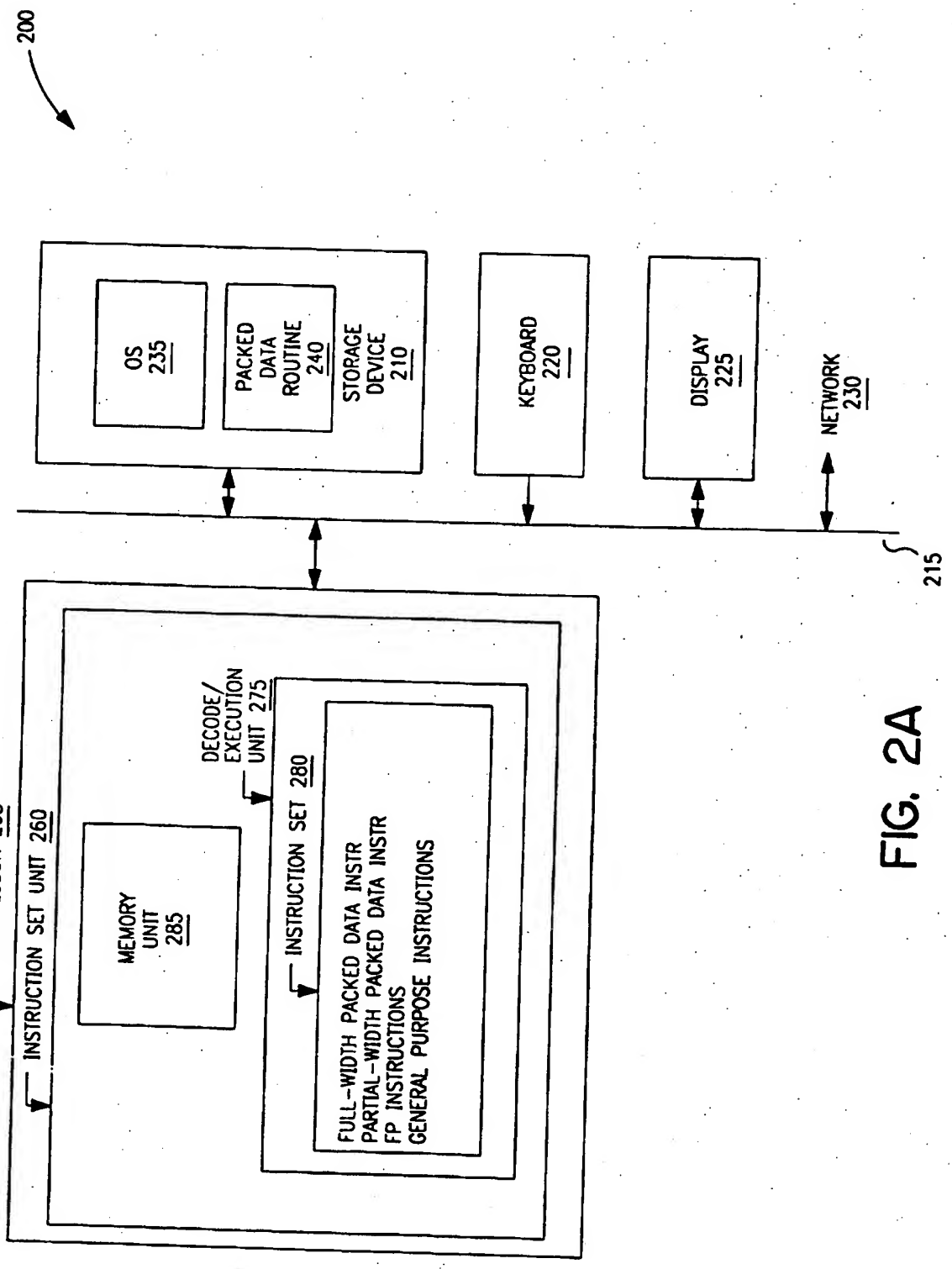

200
OS 235
PACKED DATA ROUTINE 240
STORAGE DEVICE 210
KEYBOARD 220
DISPLAY 225
NETWORK 230
215
PROCESSOR 205
INSTRUCTION SET UNIT 260
MEMORY UNIT 285
DECODE/ EXECUTION UNIT 275
INSTRUCTION SET 280
FULL-WIDTH PACKED DATA INSTR
PARTIAL-WIDTH PACKED DATA INSTR
FP INSTRUCTIONS
GENERAL PURPOSE INSTRUCTIONS

FIG. 2A

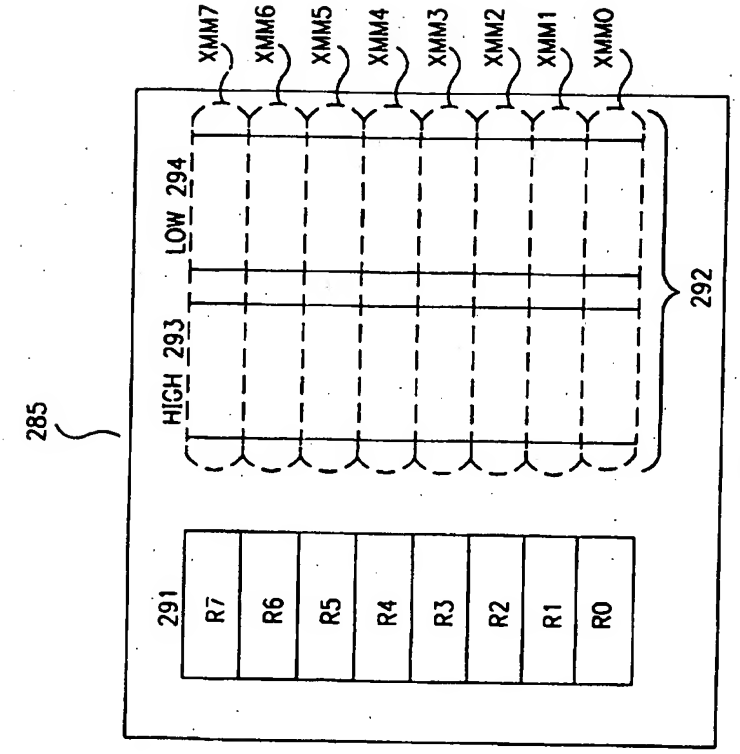

285
291
R7
R6
R5
R4
R3
R2
R1
R0
HIGH 293
LOW 294
292
XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

FIG. 2C

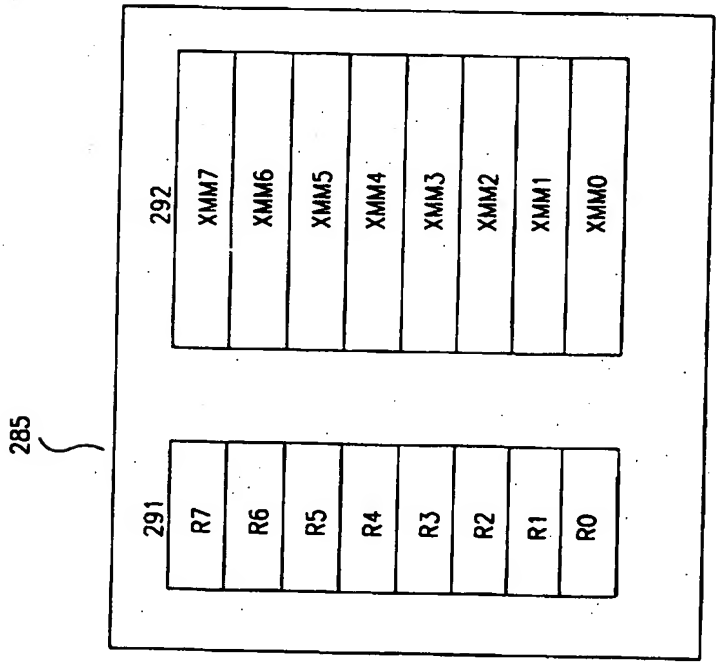

285
291
R7
R6
R5
R4
R3
R2
R1
R0
292
XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

FIG. 2B

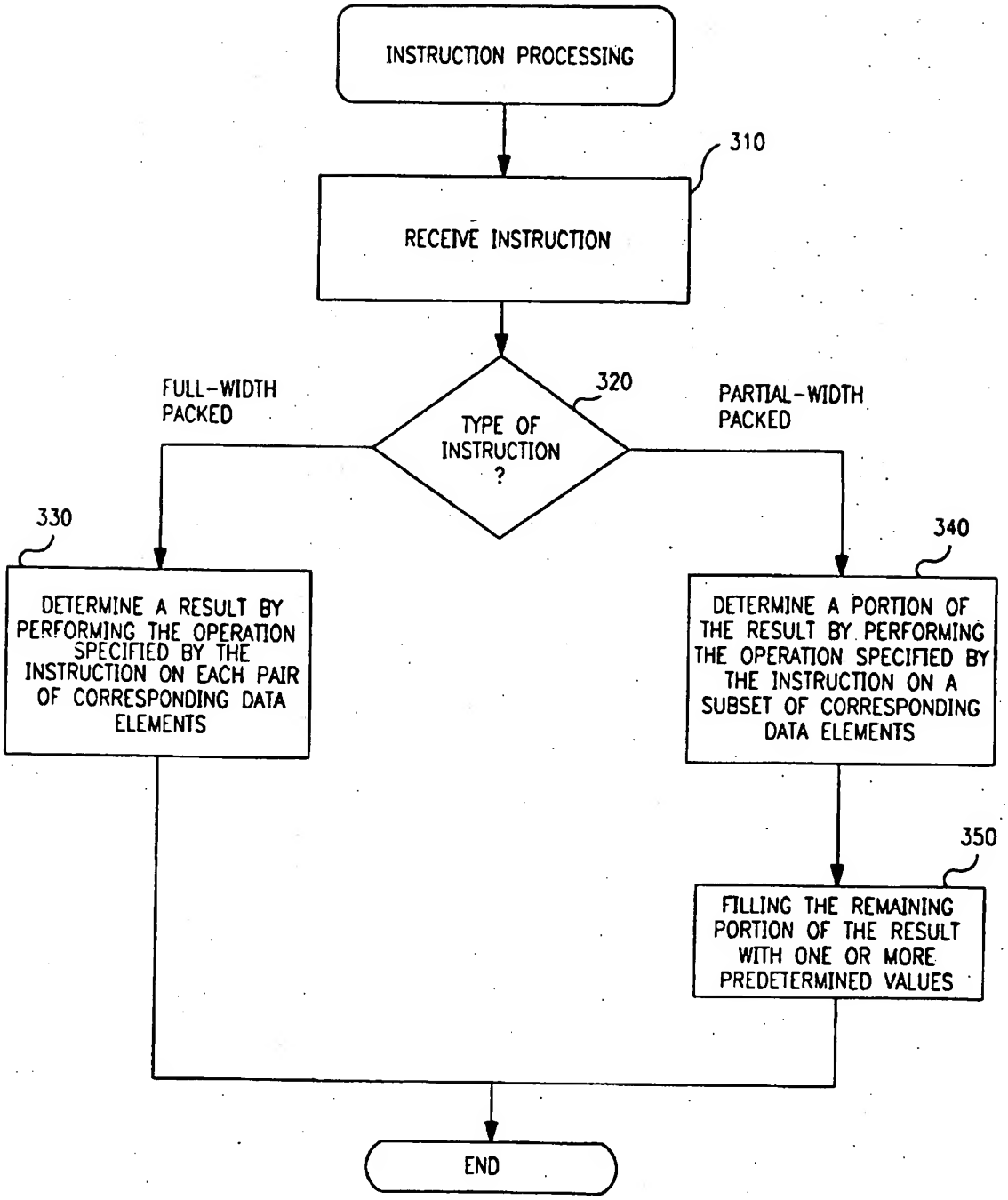

INSTRUCTION PROCESSING
310
RECEIVE INSTRUCTION
320
TYPE OF INSTRUCTION ?
FULL-WIDTH PACKED
PARTIAL-WIDTH PACKED
330
DETERMINE A RESULT BY PERFORMING THE OPERATION SPECIFIED BY THE INSTRUCTION ON EACH PAIR OF CORRESPONDING DATA ELEMENTS
340
DETERMINE A PORTION OF THE RESULT BY PERFORMING THE OPERATION SPECIFIED BY THE INSTRUCTION ON A SUBSET OF CORRESPONDING DATA ELEMENTS
350
FILLING THE REMAINING PORTION OF THE RESULT WITH ONE OR MORE PREDETERMINED VALUES
END

FIG. 3

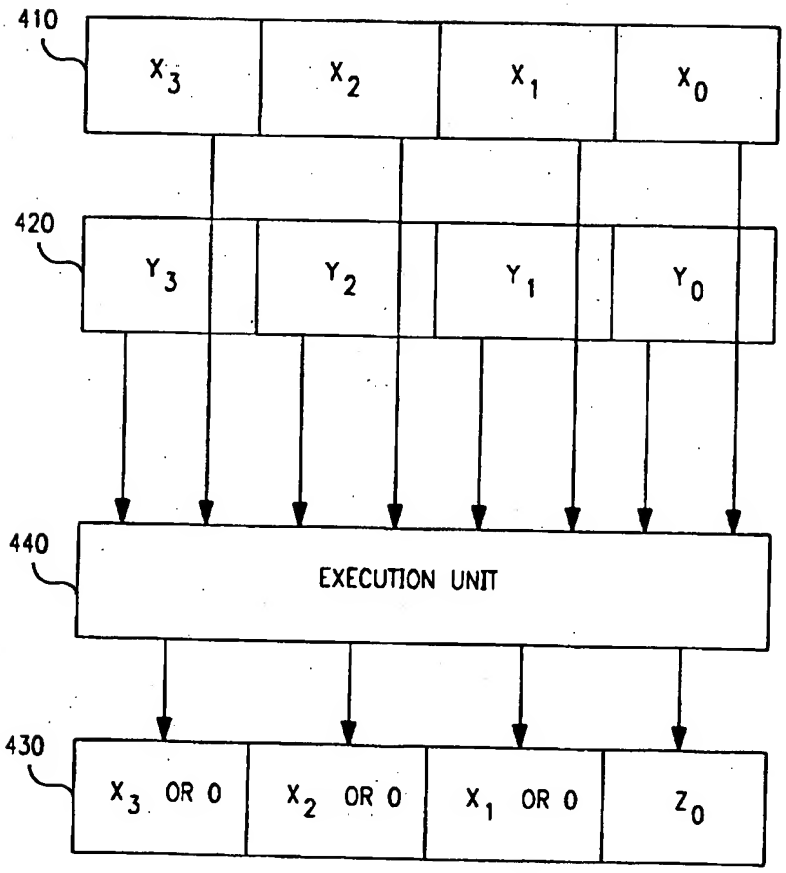

410
X3
X2
X1
X0
420
Y3
Y2
Y1
Y0
440
EXECUTION UNIT
430
X3 OR 0
X2 OR 0
X1 OR 0
Z0

FIG. 4

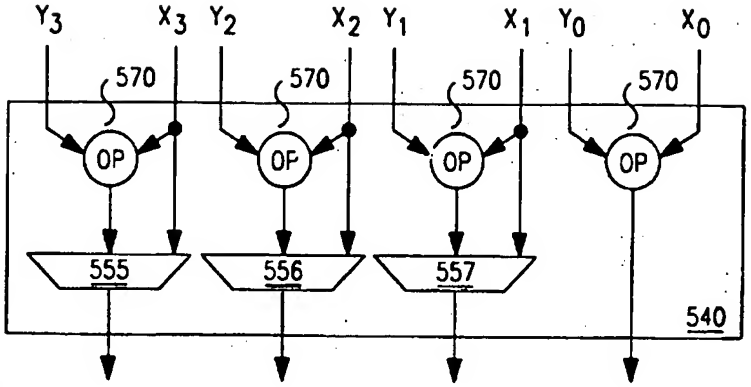


FIG. 5A

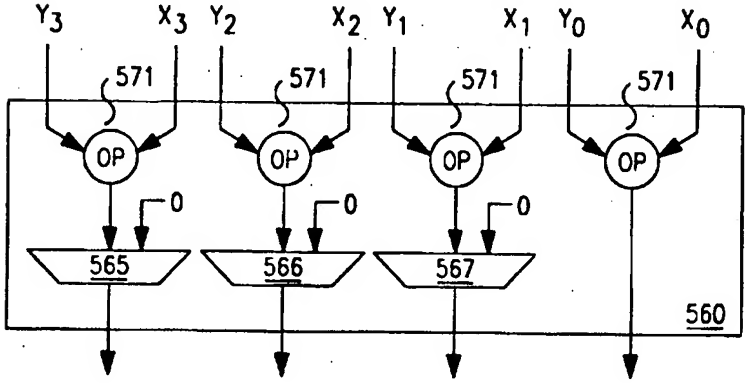


FIG. 5B

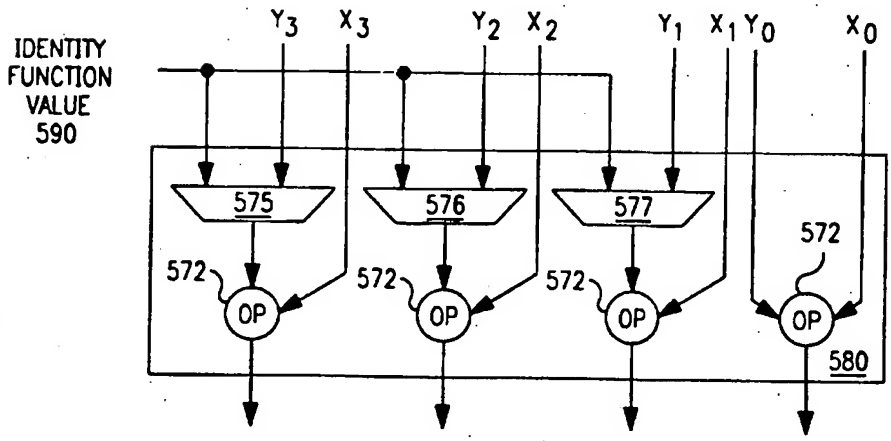


FIG. 5C

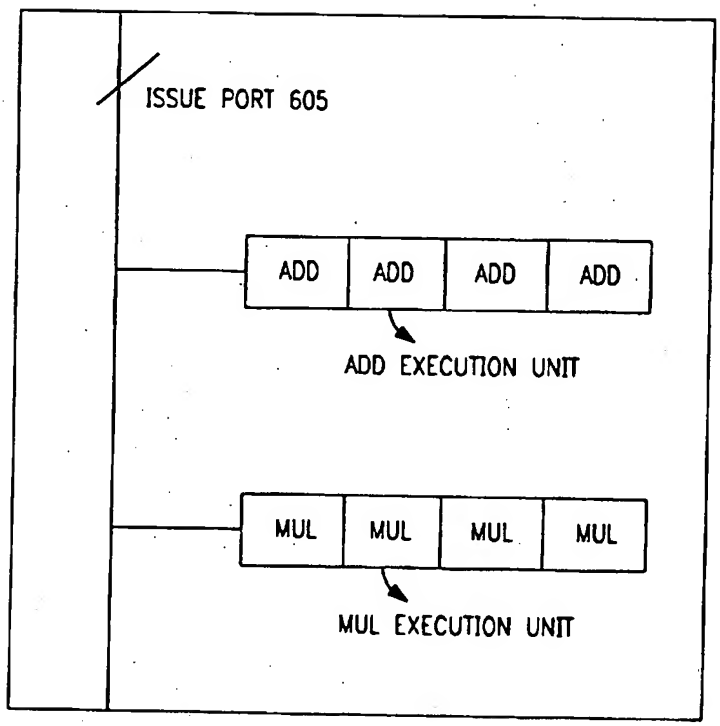

ISSUE PORT 605
ADD
ADD
ADD
ADD
ADD EXECUTION UNIT
MUL
MUL
MUL
MUL
MUL EXECUTION UNIT

FIG. 6

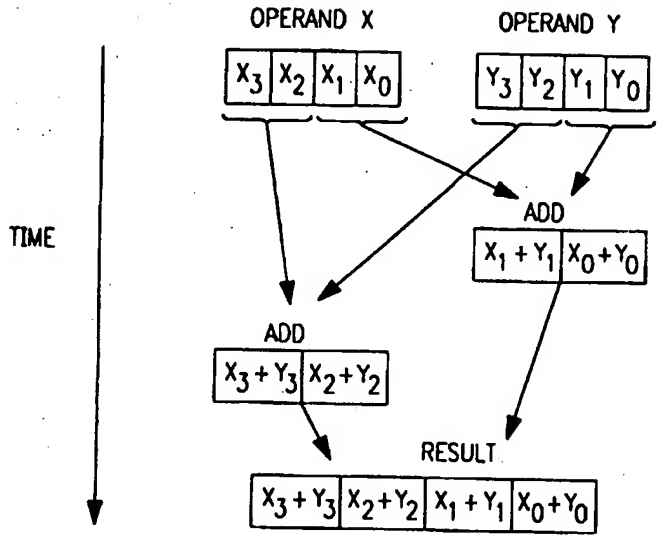
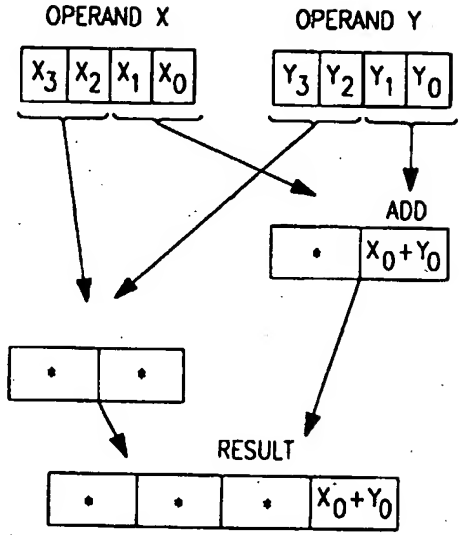


FIG. 7A



• = Corresponding data element in X or Y, NaN, Ø, or other predetermined value

FIG. 7B

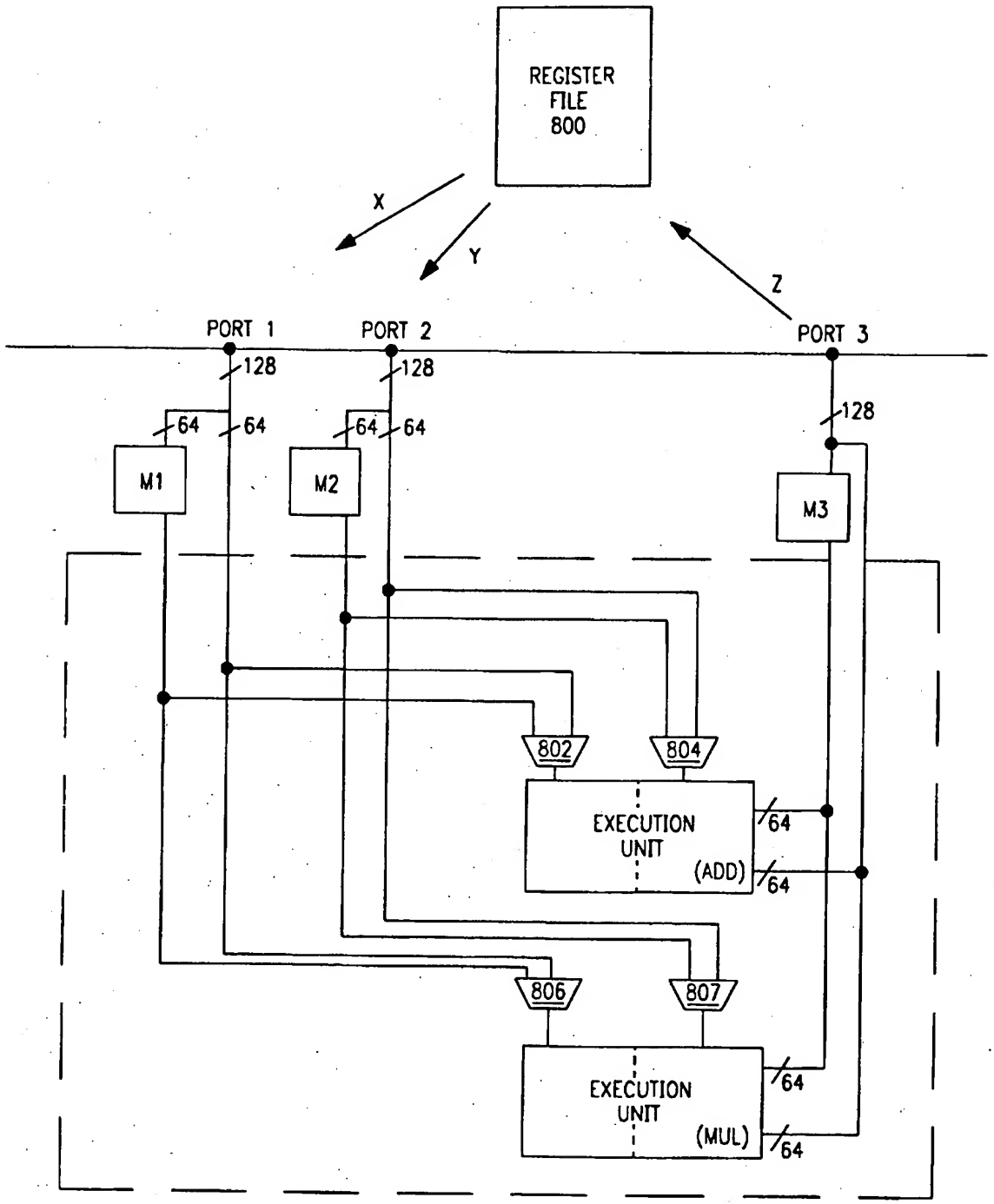

REGISTER FILE 800
X
Y
Z
PORT 1
PORT 2
PORT 3
128
128
128
64
64
64
64
M1
M2
M3
802
804
EXECUTION UNIT
(ADD)
64
64
806
807
EXECUTION UNIT
(MUL)
64
64

FIG. 8A

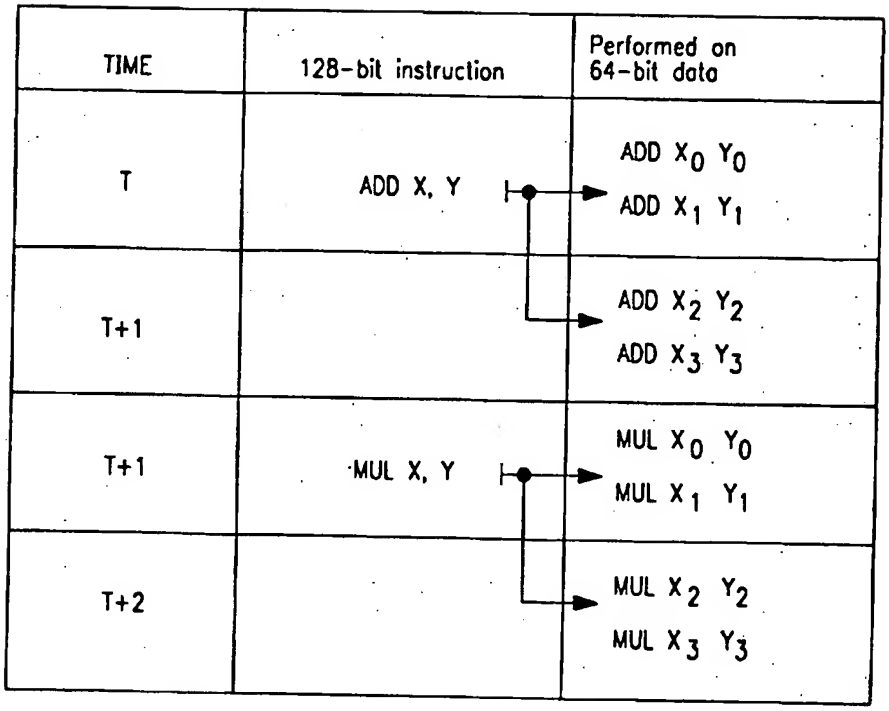

TIME
128-bit instruction
Performed on 64-bit data
T
ADD X, Y
ADD X_0 Y_0
ADD X_1 Y_1
T+1
ADD X_2 Y_2
ADD X_3 Y_3
T+1
MUL X, Y
MUL X_0 Y_0
MUL X_1 Y_1
T+2
MUL X_2 Y_2
MUL X_3 Y_3

FIG. 8B

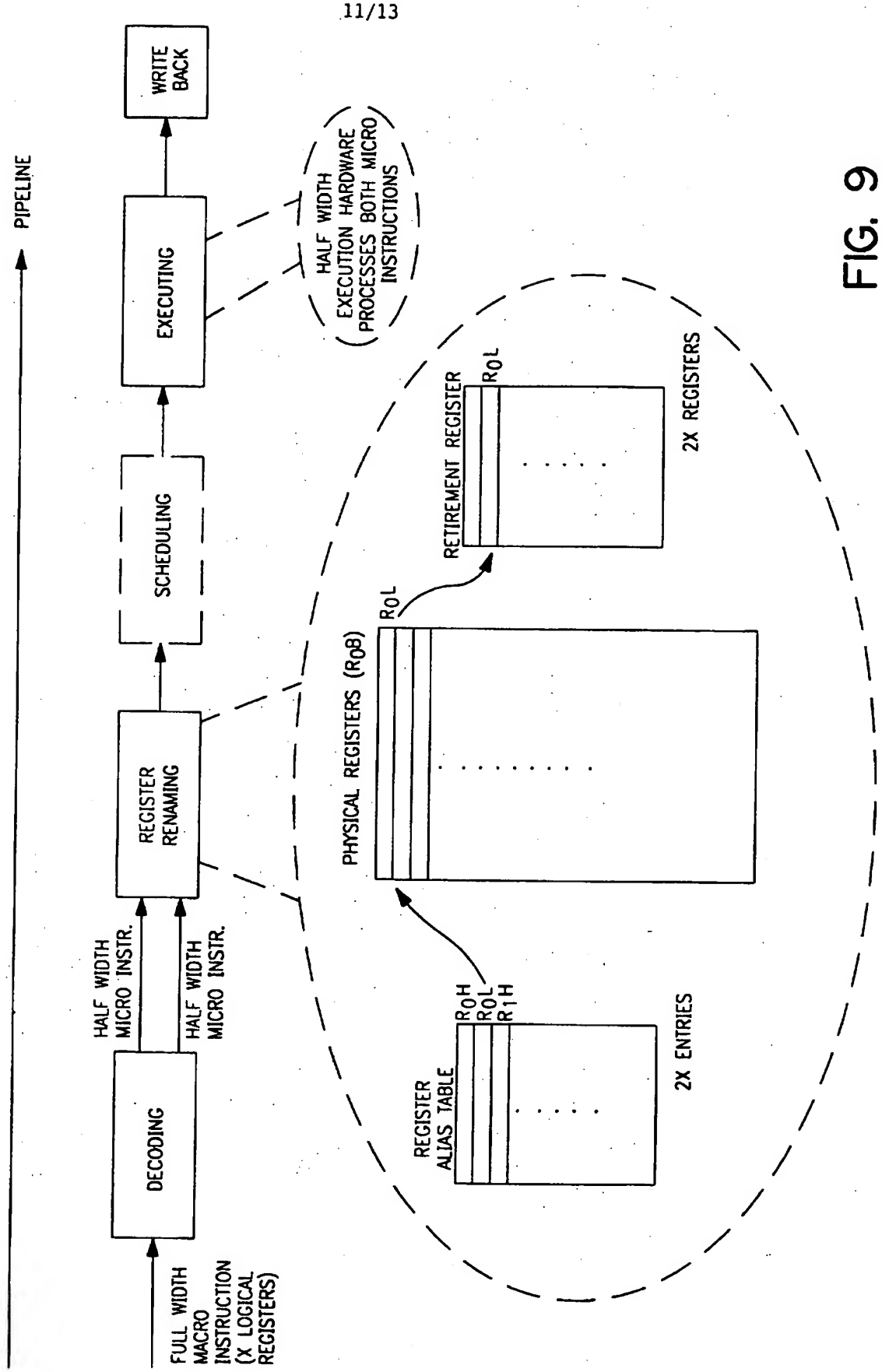

PIPELINE
FULL WIDTH MACRO INSTRUCTION (X LOGICAL REGISTERS)
DECODING
HALF WIDTH MICRO INSTR.
HALF WIDTH MICRO INSTR.
REGISTER RENAMING
SCHEDULING
EXECUTING
WRITE BACK
HALF WIDTH EXECUTION HARDWARE PROCESSES BOTH MICRO INSTRUCTIONS
REGISTER ALIAS TABLE
R0H
R0L
R1H
2X ENTRIES
PHYSICAL REGISTERS (R0-R8)
R0L
RETIREMENT REGISTER
R0L
2X REGISTERS

FIG. 9

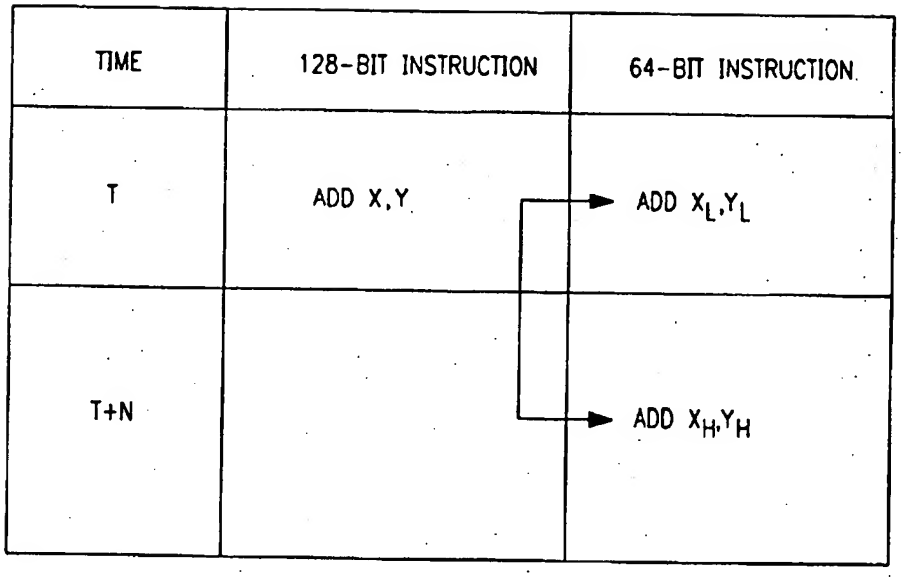

TIME
128-BIT INSTRUCTION
64-BIT INSTRUCTION
T
ADD X,Y
ADD X_L,Y_L
T+N
ADD X_H,Y_H

FIG. 10

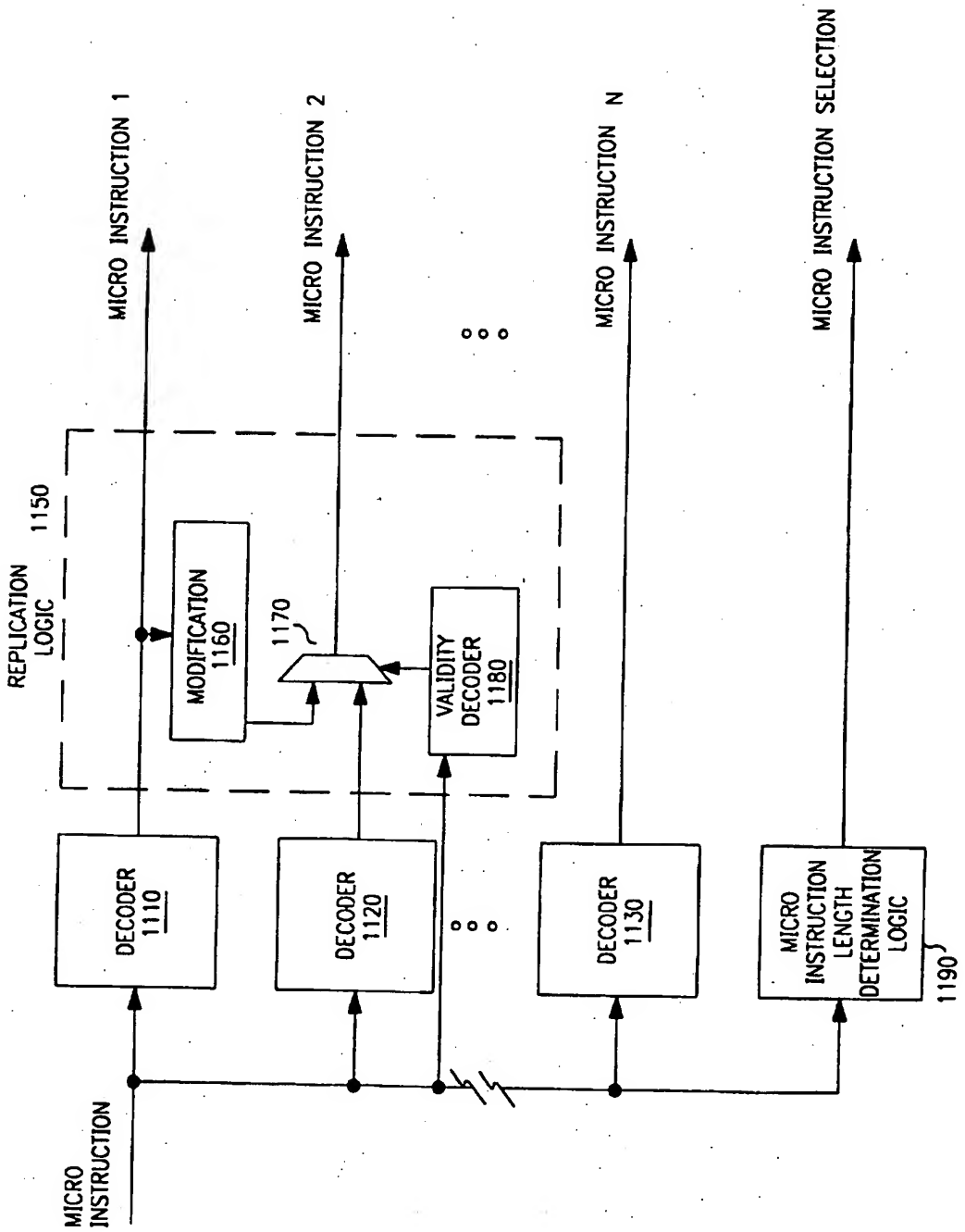

MICRO INSTRUCTION 1
MICRO INSTRUCTION 2
MICRO INSTRUCTION N
MICRO INSTRUCTION SELECTION
REPLICATION LOGIC 1150
MODIFICATION 1160
1170
VALIDITY DECODER 1180
DECODER 1110
DECODER 1120
DECODER 1130
MICRO INSTRUCTION LENGTH DETERMINATION LOGIC 1190
MICRO INSTRUCTION

FIG. 11

EXECUTING PARTIAL-WIDTH PACKED DATA INSTRUCTIONS

FIELD OF THE INVENTION

The invention relates generally to the field of computer systems. More particularly, the invention relates to a method and apparatus for efficiently executing partial-width packed data instructions, such as scalar packed data instructions, by a processor that makes use of SIMD technology, for example.

BACKGROUND OF THE INVENTION

Multimedia applications such as 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation, often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Each type of multimedia application typically implements one or more algorithms requiring a number of floating point or integer operations, such as ADD or MULTIPLY (hereafter MUL). By providing macro instructions whose execution causes a processor to perform the same operation on multiple data items in parallel, Single Instruction Multiple Data (SIMD) technology, such as that employed by the Pentium® processor architecture and the MMx™ instruction set, has enabled a significant improvement in multimedia application performance (Pentium® and MMx™ are registered trademarks or trademarks of Intel Corporation of Santa Clara, CA).

SIMD technology is especially suited to systems that provide packed data formats. A packed data format is one in which the bits in a register are logically divided into a number of fixed-sized data elements, each of which represents a separate value. For example, a 64-bit register may be broken into four 16-bit elements, each of which represents a separate 16-bit value. Packed data instructions may then separately manipulate each element in these packed data types in parallel.

Referring to **Figure 1**, an exemplary packed data instruction is illustrated. In this example, a packed ADD instruction (e.g., a SIMD ADD) adds corresponding data elements of a first packed data operand, X, and a second packed data operand, Y, to produce a packed data result, Z, i.e., $X_0+Y_0=Z_0$, $X_1+Y_1=Z_1$, $X_2+Y_2=Z_2$, and $X_3+Y_3=Z_3$. Packing many data elements within one register or memory location and employing parallel hardware execution allows SIMD architectures to perform multiple operations at a time, resulting in significant performance improvement. For instance, in this example, four individual results may be obtained in the time previously required to obtain a single result.

While the advantages achieved by SIMD architectures are evident, there remain situations in which it is desirable to return individual results for only a subset of the packed data elements.

SUMMARY OF THE INVENTION

A method and apparatus are described for executing partial-width packed data instructions. According to one aspect of the invention, a processor includes a plurality of registers, a register renaming unit coupled to the plurality of registers, a decoder coupled to the register renaming unit, and a partial-width execution unit coupled to the decoder. The register renaming unit provides an architectural register file to store packed data operands each of which include a plurality of data elements. The decoder is configured to decode a first and second set of instructions that each specify one or more registers in the architectural register file. Each of the instructions in the first set of instructions specify operations to be performed on all of the data elements stored in the one or more specified registers. In contrast, each of the instructions in the second set of instructions specify operations to be performed on only a subset of the data element stored in the one or more specified registers. The partial-width execution unit is configured to execute operations specified by either of the first or the second set of instructions.

Other features and advantages of the invention will be apparent from the accompanying drawings and from the detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is described by way of example and not by way of limitation with reference to the figures of the accompanying drawings in which like reference numerals refer to similar elements and in which:

Figure 1 illustrates a packed ADD instruction adding together corresponding data elements from a first packed data operand and a second packed data operand.

Figure 2A is a simplified block diagram illustrating an exemplary computer system according to one embodiment of the invention.

Figure 2B is a simplified block diagram illustrating exemplary sets of logical registers according to one embodiment of the invention.

Figure 2C is a simplified block diagram illustrating exemplary sets of logical registers according to another embodiment of the invention.

Figure 3 is a flow diagram illustrating instruction execution according to one embodiment of the invention.

Figure 4 conceptually illustrates the result of executing a partial-width packed data instruction according to various embodiments of the invention.

Figure 5A conceptually illustrates circuitry for executing full-width packed data instructions and partial-width packed data instructions according to one embodiment of the invention.

Figure 5B conceptually illustrates circuitry for executing full-width packed data and partial-width packed data instructions according to another embodiment of the invention.

Figure 5C conceptually illustrates circuitry for executing full-width packed data and partial-width packed data instructions according to yet another embodiment of the invention.

Figure 6 illustrates an ADD execution unit and a MUL execution unit capable of operating as four separate ADD execution units and four separate MUL execution units, respectively, according to an exemplary processor implementation of SIMD.

Figures 7A-7B conceptually illustrate a full-width packed data operation and a partial-width packed data operation being performed in a "staggered" manner, respectively.

Figure 8A conceptually illustrates circuitry within a processor that accesses full width operands from logical registers while performing operations on half of the width of the operands at a time.

Figure 8B is a timing chart that further illustrates the circuitry of Figure 8A.

Figure 9 conceptually illustrates one embodiment of an out-of-order pipeline to perform operations on operands in a "staggered" manner by converting a macro instruction into a plurality of micro instructions that each processes a portion of the full width of the operands.

Figure 10 is a timing chart that further illustrates the embodiment described in Figure 9.

Figure 11 is a block diagram illustrating decoding logic that may be employed to accomplish the decoding processing according to one embodiment of the invention.

DETAILED DESCRIPTION

A method and apparatus are described for performing partial-width packed data instructions. Herein the term "full-width packed data instruction" is meant to refer to a packed data instruction (e.g., a SIMD instruction) that operates upon all of the data elements of one or more packed data operands. In contrast, the term "partial-width packed data instruction" is meant to broadly refer to a packed data instruction that is designed to operate upon only a subset of the data elements of one or more packed data operands and return a packed data result (to a packed data register file, for example). For instance, a scalar SIMD instruction may require only a result of an operation between the least significant pair of packed data operands. In this example, the remaining data elements of the packed data result are disregarded as they are of no consequence to the scalar SIMD instruction (e.g., the remaining data elements are don't cares). According to the various embodiments of the invention, execution units may be configured in such a way to efficiently accommodate both full-width packed data instructions (e.g., SIMD instructions) and a set of partial-width packed data instructions (e.g., scalar SIMD instructions).

In the following detailed description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one of ordinary skill in the art that these specific details need not be used to practice

the invention. In other instances, well-known devices, structures, interfaces, and processes have not been shown or are shown in block diagram form.

Justification of Partial-Width Packed Data Instructions

Considering the amount of software that has been written for scalar architectures (e.g., single instruction single data (SISD) architectures) employing scalar operations on single precision floating point data, double precision floating point data, and integer data, it is desirable to provide developers with the option of porting their software to architectures that support packed data instructions, such as SIMD architectures, without having to rewrite their software and/or learn new instructions. By providing partial-width packed data instructions, a simple translation can transform old scalar code into scalar packed data code. For example, it would be very easy for a compiler to produce scalar SIMD instructions from scalar code. Then, as developers recognize portions of their software that can be optimized using SIMD instructions, they may gradually take advantage of the packed data instructions. Of course, computer systems employing SIMD technology are likely to also remain backwards compatible by supporting SISD instructions as well. However, the many recent architectural improvements and other factors discussed herein make it advantageous for developers to transition to and exploit SIMD technology, even if only scalar SIMD instructions are employed at first.

Another justification for providing partial-width packed data instructions is the many benefits which may be achieved by operating on only a subset of a full-width operand, including reduced power consumption, increased speed, a clean exception model, and increased storage. As illustrated below, based on an indication provided with the partial-width packed data instruction, power savings may be achieved by selectively shutting down those of the hardware units that are unnecessary for performing the current operation.

Another situation in which it is undesirable to force a packed data instruction to return individual results for each pair of data elements includes arithmetic operations in an environment providing partial-width hardware. Due to cost and/or die limitations, it is common not to provide full support for certain arithmetic operations, such as divide. the By its nature, the divide operation is very long, even when full-width hardware (e.g., a one-to-one correspondence between execution units and data elements) is implemented,. Therefore, in an environment that supports only full-width packed data operations while providing partial-width hardware, the latency becomes even longer. As will be illustrated further below, a partial-width packed data operation, such as a partial-width packed data divide operation, may selectively allow certain portions of its operands to bypass the divide hardware. In this manner, no performance penalty is incurred by operating upon only a subset of the data elements in the packed data operands.

Additionally, exceptions raised in connection with extraneous data elements may cause confusion to the developer and/or incompatibility between SISD and SIMD machines. Therefore, it is advantageous to report exceptions for only those data elements upon which the instruction is meant to operate. Partial-width packed data instruction support allows a predictable exception model to be achieved by limiting the triggering of exceptional conditions to those raised in connection with the data elements being operated upon. or in which exceptions produced by extraneous data elements would be likely to cause confusion or incompatibility between SISD and SIMD machines.

Finally, in embodiments where portions of destination packed data operand is not corrupted as a result of performing a partial-width packed data operation, partial-width packed data instructions effectively provide extra register space for storing data. For instance, if the lower portion of the packed data operand is being operated upon, data may be stored in the upper portion and vice versa.

An Exemplary Computer System

Figure 2A is a simplified block diagram illustrating an exemplary computer system according to one embodiment of the invention. In the embodiment depicted, computer system 200 includes a processor 205, a storage device 210, and a bus 215. The processor 205 is coupled to the storage device 210 by the bus 215. In addition, a number of user input/output devices, such

as a keyboard 220 and a display 225 are also coupled to bus 215. The computer system 200 may also be coupled to a network 230 via bus 215. The processor 205 represents a central processing unit of any type of architecture, such as a CISC, RISC, VLIW, or hybrid architecture. In addition, the processor 205 may be implemented on one or more chips. The storage device 210 represents one or more mechanisms for storing data. For example, the storage device 210 may include read only memory (ROM), random access memory (RAM), magnetic disk storage mediums, optical storage mediums, flash memory devices, and/or other machine-readable mediums. The bus 215 represents one or more buses (e.g., AGP, PCI, ISA, X-Bus, EISA, VESA, etc.) and bridges (also termed as bus controllers). While this embodiment is described in relation to a single processor computer system, it is appreciated that the invention may be implemented in a multi-processor computer system. In addition while the present embodiment is described in relation to a 32-bit and a 64-bit computer system, the invention is not limited to such computer systems.

Figure 2A additionally illustrates that the processor 205 includes an instruction set unit 260. Of course, processor 205 contains additional circuitry; however, such additional circuitry is not necessary to understanding the invention. An any rate, the instruction set unit 260 includes the hardware and/or firmware to decode and execute one or more instruction sets. In the embodiment depicted, the instruction set unit 260 includes a decode/execution unit 275. The decode unit decodes instructions received by

processor 205 into one or more micro instructions. The execution unit performs appropriate operations in response to the micro instructions received from the decode unit. The decode unit may be implemented using a number of different mechanisms (e.g., a look-up table, a hardware implementation, a PLA, etc.).

In the present example, the decode/execution unit 275 is shown containing an instruction set 280 that includes both full-width packed data instructions and partial-width packed data instructions. These packed data instructions, when executed, may cause the processor 205 to perform full-/partial-width packed floating point operations and/or full-/partial-width packed integer operations. In addition to the packed data instructions, the instruction set 280 may include other instructions found in existing micro processors. By way of example, in one embodiment the processor 205 supports an instruction set which is compatible with Intel 32-bit architecture (IA-32) and/or Intel 64-bit architecture (IA-64).

A memory unit 285 is also included in the instruction set unit 260. The memory unit 285 may include one or more sets of architectural registers (also referred to as logical registers) utilized by the processor 205 for storing information including floating point data and packed floating point data. Additionally, other logical registers may be included for storing integer data, packed integer data, and various control data, such as a top of stack indication and the like. The terms architectural register and logical register are used herein to refer to the concept of the manner in which instructions specify a

storage area that contains a single operand. Thus, a logical register may be implemented in hardware using any number of well known techniques, including a dedicated physical register, one or more dynamically allocated physical registers using a register renaming mechanism (described in further detail below), etc.. In any event, a logical register represents the smallest unit of storage addressable by a packed data instruction.

In the embodiment depicted, the storage device 210 has stored therein an operating system 235 and a packed data routine 240 for execution by the computer system 200. The packed data routine 240 is a sequence of instructions that may include one or more packed data instructions, such as scalar SIMD instructions or SIMD instructions. As discussed further below, there are situations, including speed, power consumption and exception handling, where it is desirable to perform an operation on (or return individual results for) only a subset of data elements in a packed data operand or a pair of packed data operands. Therefore, it is advantageous for processor 205 to be able to differentiate between full-width packed data instructions and partial-width packed data instructions and to execute them accordingly.

Figure 2B is a simplified block diagram illustrating exemplary sets of logical registers according to one embodiment of the invention. In this example, the memory unit 285 includes a plurality of scalar floating point registers 291 (a scalar register file) and a plurality of packed floating point registers 292 (a packed data register file). The scalar floating point registers 291 (e.g., registers R_0-R_7) may be implemented as a stack referenced register file

when floating point instructions are executed so as to be compatible with existing software written for the Intel Architecture. In alternative embodiments, however, the registers 291 may be treated as a flat register file. In the embodiment depicted, each of the packed floating point registers (e.g., XMM0-XMM7) are implemented as a single 128-bit logical register. It is appreciated, however, wider or narrower registers may be employed to conform to an implementation that uses more or less data elements or larger or smaller data elements. Additionally, more or less packed floating point registers 292 may be provided. Similar to the scalar floating point registers 291, the packed floating point registers 292 may be implemented as either a stack referenced register file or a flat register file when packed floating point instructions are executed.

Figure 2C is a simplified block diagram illustrating exemplary sets of logical registers according to another embodiment of the invention. In this example, the memory unit 285, again, includes a plurality of scalar floating point registers 291 (a scalar register file) and a plurality of packed floating point registers 292 (a packed data register file). However, in the embodiment depicted, each of the packed floating point registers (e.g., XMM0-XMM7) are implemented as a corresponding pairs of high 293 and low registers 294. As will be discussed further below, it is advantageous for purposes of instruction decoding to organize the logical register address space for the packed floating point registers 292 such that the high and low register pairs differ by a single bit. For example, the high and low portions of XMM0-XMM7 may be

differentiated by the MSB. Preferably, each of the packed floating point registers 291 are wide enough to accommodate four 32-bit single precision floating point data elements. As above, however, wider or narrower registers may be employed to conform to an implementation that uses more or less data elements or larger or smaller data elements. Additionally, while the logical packed floating point registers 292 in this example each comprise corresponding pairs of 64-bit registers, in alternative embodiments each packed floating point register may comprise any number of registers.

Instruction Execution Overview

Having described an exemplary computer system in which one embodiment of the invention may be implemented, instruction execution will now be described.

Figure 3 is a flow diagram illustrating instruction execution according to one embodiment of the invention. At step 310, an instruction is received by the processor 205. At step 320, based on the type of instruction, partial-width packed data instruction (e.g., scalar SIMD instruction) or full-width packed data instruction (e.g., SIMD instruction), processing continues with step 330 or step 340. Typically, in the decode unit the type of instruction is determined based on information contained within the instruction. For example, information may be included in a prefix or suffix that is appended to an opcode or provided via an immediate value to indicate whether the corresponding operation is to be performed on all or a subset of the data

elements of the packed data operand(s). In this manner, the same opcodes may be used for both full-width packed data operations and partial-width packed data operations. Alternatively, one set of opcodes may be used for partial-width packed data operations and a different set of opcodes may be used for full-width packed data operations.

In any event, if the instruction is a conventional full-width packed data instruction, then at step 330, a packed data result is determined by performing the operation specified by the instruction on each of the data elements in the operand(s). However, if the instruction is a partial-width packed data instruction, then at step 340, a first portion of the result is determined by performing the operation specified by the instruction on a subset of the data elements and the remainder of the result is set to one or more predetermined values. In one embodiment, the predetermined value is the value of the corresponding data element in one of the operands. That is, data elements may be "passed through" from data elements of one of the operands to corresponding data elements in the packed data result. In another embodiment, the data elements in the remaining portion of the result are all cleared (zeroed). Exemplary logic for performing the passing through of data elements from one of the operands to the result and exemplary logic for clearing data elements in the result are described below.

Figure 4 conceptually illustrates the result of executing a partial-width packed data instruction according to various embodiments of the invention. In this example, an operation is performed on data elements of two logical

source registers 410 and 420 by an execution unit 440. The execution unit 440 includes circuitry and logic for performing the operation specified by the instruction. In addition, the execution unit 440 may include selection circuitry that allows the execution unit 440 to operate in a partial-width packed data mode or a full-width packed data mode. For instance, the execution unit 440 may include pass through circuitry to pass data elements from one of the logical source registers 410, 420 to the logical destination register 430, or clearing circuitry to clear one or more data elements of the logical destination register 430, etc. Various other techniques may also be employed to affect the result of the operation, including forcing one of the inputs to the operation to a predetermined value, such as a value that would cause the operation to perform its identity function or a value that may pass through arithmetic operations without signaling an exception (e.g., a quiet not-a-number (QNaN)).

In the example illustrated, only the result (Z_0) of the operation on the first pair of data elements (X_0 and Y_0) is stored in the logical destination register 430. Assuming the execution unit 440 includes pass through logic, the remaining data elements of the logical destination register 430 are set to values from corresponding data elements of logical source register 410 (i.e., X_3, X_2, and X_1). While the logical destination register 430 is shown as a separate logical register, it is important to note that it may concurrently serve as one of the logical source registers 410, 420. Therefore, it should be appreciated that setting data elements of the logical destination register 430 to

values from one of the logical source registers 410, 420 in this context may include doing nothing at all. For example, in the case that logical source register 410 is both a logical source and destination register, various embodiments may take advantage of this and simply not touch one or more of the data elements which are to be passed through.

Alternatively, the execution unit 440 may include clearing logic. Thus, rather than passing through values from one of the logical source registers to the logical destination register 430, those of the data elements in the result that are unnecessary are cleared. Again, in this example, only the result (Z_0) of the operation on the first pair of data elements (X_0 and Y_0) is stored in the logical destination register 430. The remaining data elements of the logical destination register 430 are "cleared" (e.g., set to zero, or any other predetermined value for that matter).

Full-Width Hardware

Figures 5A - 5C conceptually illustrate execution units 540, 560 and 580, respectively, which may execute both full-width packed data and partial-width packed data instructions. The selection logic included in the execution units of Figures 5A and 5C represent exemplary pass through logic, while the selection logic of Figure 5B is representative of clearing logic that may be employed. In the embodiments depicted, the execution units 540, 560, and 580 each include appropriate logic, circuitry and/or firmware for concurrently

performing an operation 570, 571, and 572 on the full-width of the operands (X and Y).

Referring now to Figure 5A, the execution unit 540 includes selection logic (e.g., multiplexers (MUXes) 555-557) for selecting between a value produced by the operation 570 and a value from a corresponding data element of one of the operands. The MUXes 555-557 may be controlled, for example, by a signal that indicates whether the operation currently being executed is a full-width packed data operation or a partial-width packed data operation. In alternative embodiments, additional flexibility may be achieved by including an additional MUX for data element 0 and/or independently controlling each MUX. Various means of providing MUX control are possible. According to one embodiment, such control may originate or be derived from the instruction itself or may be provided via immediate values. For example, a 4-bit immediate value associated with the instruction may be used to allow the MUXes 555-557 to be controlled directly by software. Those MUXes corresponding to a one in the immediate value may be directed to select the result of the operation while those corresponding to a zero may be caused to select the pass through data. Of course, more or less resolution may be achieved in various implementations by employing more or less bits to represent the immediate value.

Turning now to Figure 5B, the execution unit 540 includes selection logic (e.g., MUXes 565-567) for selecting between a value produced by an

operation 571 and a predetermined value (e.g., zero). As above, the MUXes 565-567 may be under common control or independently controlled.

The pass through logic of Figure 5C (e.g., MUXes 575-576) selects between a data element of one of the operands and an identity function value 590. The identity function value 590 is generally chosen such that the result of performing the operation 572 between the identity function value 590 and the data element is the value of the data element. For example, if the operation 572 was a multiply operation, then the identity function value 590 would be 1. Similarly, if the operation 572 was an add operation, the identity function value 590 would be 0. In this manner, the value of a data element can be selectively passed through to the logical destination register 430 by causing the corresponding MUX 575-577 to output the identity function value 590.

In the embodiments described above, the circuitry was hardwired such that the partial-width operation was performed on the least significant data element portion. It is appreciated that the operation may be performed on a different data element portions than illustrated. Also, as described above, the data elements to be operated upon may be made to be software configurable by coupling all of the operations to a MUX or the like, rather than simply a subset of the operations as depicted in Figures 5A-5C. Further, while pass through and clearing logic are described as two options for treating resulting data elements corresponding to operations that are to be disregarded, alternative embodiments may employ other techniques. For example, a

QNaN may be input as one of the operands to an operation whose result is to be disregarded. In this manner, arithmetic operations compliant with the IEEE 754 standard, IEEE std. 754-1985, published March 21, 1985, will propagate a NaN through to the result without triggering an arithmetic exception.

While no apparent speed up would be achieved in the embodiments described above since the full-width of the operands can be processed in parallel, it should be appreciated that power consumption can be reduced by shutting down those of the operations whose results will be disregarded. Thus, significant power savings may be achieved. Additionally, with the use of QNaNs and/or identity function values a predictable exception model may be maintained by preventing exceptions from being triggered by data elements that are not part of the partial-width packed data operation. Therefore, reported exceptions are limited to those raised in connection with the data element(s) upon which the partial-width packed data operation purports to operate.

Figure 6 illustrates a current processor implementation of an arithmetic logic unit (ALU) that can be used to execute full-width packed data instructions. The ALU of Figure 6 includes the circuitry necessary to perform operations on the full width of the operands (i.e., all of the data elements). Figure 6 also shows that the ALU may contain one or more different types of execution units. In this example, the ALU includes two different types of execution units for respectively performing different types of operations (e.g., certain ALUs use separate units for performing ADD and MUL operations).

The ADD execution unit and the MUL execution unit are respectively capable of operating as four separate ADD execution units and four separate MUL execution units. Alternatively, the ALU may contain one or more Multiply Accumulate (MAC) units, each capable of performing more than a single type of operation. While the following examples assume the use of ADD and MUL execution units and floating point operations, it is appreciated other execution units such as MAC and/or integer operations may also be used. Further, it may be preferable to employ a partial-width implementation (e.g., an implementation with less than a one-to-one correspondence between execution units and data elements) and additional logic to coordinate reuse of the execution units as described below.

Partial-Width Hardware and "Staggered Execution"

Figures 7A-7B conceptually illustrate a full-width packed data operation and a partial-width packed data operation being performed in a "staggered" manner, respectively. "Staggered execution" in the context of this embodiment refers to the process of dividing each of an instruction's operands into separate segments and sequentially processing each segment using the same hardware. The segments are sequentially processed by introducing a delay into the processing of the subsequent segments. As illustrated in Figures 7A-7B, in both cases, the packed data operands are divided into a "high order segment" (data elements 3 and 2) and a "low order segment" (data elements 1 and 0). In the example of Figure 7A, the low order

segment is processed while the high order segment is delayed. Subsequently, the high order segment is processed and the full-width result is obtained. In the example of Figure 7B, the low order segment is processed, while whether the high order data segment is processed depends on the implementation. For example, the high order data segment may not need to be processed if the corresponding result is to be zeroed. Additionally, it is appreciated that if the high order data segment is not processed, then both the high and low order data segments may be operated upon at the same time. Similarly, in a full-width implementation (e.g., an implementation with a one-to-one correspondence between execution units and data elements) the high and low order data segments may be processed concurrently or as shown in Figure 7A.

Additionally, although the following embodiments are described as having only ADD and MUL execution units, other types of execution units such as MAC units may also be used.

While there are a number of different ways in which the staggered execution of instructions can be achieved, the following sections describe two exemplary embodiments to illustrate this aspect of the invention. In particular, both of the described exemplary embodiments receive the same macro instructions specifying logical registers containing 128 bit operands.

In the first exemplary embodiment, each macro instruction specifying logical registers containing 128 bit operands causes the full-width of the operands to be accessed from the physical registers. Subsequent to accessing the full-width operands from the registers, the operands are divided into the

low and high order segments (e.g., using latches and multiplexers) and sequentially executed using the same hardware. The resulting half-width results are collected and simultaneously written to a single logical register.

In contrast, in the second exemplary embodiment each macro instruction specifying logical registers containing 128 bit operands is divided into at least two micro instructions that each operate on only half of the operands. Thus, the operands are divided into a high and low order segment and each micro instruction separately causes only half of the operands to be accessed from the registers. This type of a division is possible in a SIMD architecture because each of the operands is independent from the other. While implementations of the second embodiment can execute the micro instructions in any order (either an in order or an out of order execution model), the micro instructions respectively cause the operation specified by the macro instruction to be independently or separately performed on the low and high order segments of the operands. In addition, each micro instruction causes half of the resulting operand to be written into the single destination logical register specified by the macro instruction.

While embodiments are described in which 128 bit operands are divided into two segments, alternative embodiments could use larger or smaller operands and/or divide those operands into more than two segments. In addition, while two exemplary embodiments are described for performing staggered execution, alternative embodiments could use other techniques.

First Exemplary Embodiment Employing "Staggered Execution"

Figure 8A conceptually illustrates circuitry within a processor according to a first embodiment that accesses full width operands from the logical registers but that performs operations on half of the width of the operands at a time. This embodiment assumes that the processor execution engine is capable of processing one instruction per clock cycle. By way of example, assume the following sequence of instructions is executed: ADD X, Y; MUL A, B. At time T, 128-bits of X and 128-bits of Y are each retrieved from their respective physical registers via ports 1 and 2. The lower order data segments, namely the lower 64 bits, of both X and Y are passed into multiplexers 802 and 804 and then on to the execution units for processing. The higher order data segments, the higher 64 bits of X and Y are held in delay elements M1 and M2. At time T+1, the higher order data segments of X and Y are read from delay elements M1 and M2 and passed into multiplexers 802 and 804 and then on to the execution units for processing. In general, the delay mechanism of storing the higher order data segments in delay elements M1 and M2 allows N-bit (N=64 in this example) hardware to process 2N-bits of data. The low order results from the execution unit are then held in delay element M3 until the high order results are ready. The results of both processing steps are then written back to register file 800 via port 3. Recall that in the case of a partial-width packed data operation one or more data elements of the low or high order results may be forced to a predetermined value (e.g., zero, the value of a

corresponding data element in one of X or Y, etc.) rather than the output of the ADD or MUL operation.

Continuing with the present example, at time T+1, the MUL instruction may also have been started. Thus, at time T+1, 128-bits of A and B may each have been retrieved from their respective registers via ports 1 and 2. The lower order data segments, namely the lower 64-bits, of both A and B may be passed into multiplexers 806 and 808. After the higher order bits of X and Y are removed from delay elements M1 and M2 and passed into multiplexers 806 and 808, the higher order bits of A and B may be held in storage in delay elements M1 and M2. The results of both processing steps is written back to register file 800 via port 3.

Thus, according to an embodiment of the invention, execution units are provided that contain only half the hardware (e.g. two single precision ADD execution units and two single precision MUL execution units), instead of the execution units required to process the full width of the operands in parallel as found in a current processor. This embodiment takes advantage of statistical analysis showing that multimedia applications utilize approximately fifty percent ADD instructions and fifty percent MUL instructions. Based on these statistics, this embodiment assumes that multimedia instructions generally follow the following pattern: ADD, MUL, ADD, MUL, etc.. By utilizing the ADD and MUL execution units in the manner described above, the present embodiment provides for an optimized

use of the execution units, thus enabling comparable performance to the current processor, but at a lower cost.

Figure 8B is a timing chart that further illustrates the circuitry of Figure 8A. More specifically, as illustrated in Figure 8B, when instruction "ADD X, Y" is issued at time T, the two ADD execution units first perform ADDs on the lower order data segments or the lower two packed data elements of Figure 1, namely X_0Y_0 and X_1Y_1. At time T + 1, the ADD operation is performed on the remaining two data elements from the operands, by the same execution units, and the subsequent two data elements of the higher order data segment are added, namely X_2Y_2 and X_3Y_3. While the above embodiment is described with reference to ADD and MUL operations using two execution units, alternate embodiments may use any number of execution units and/or execute any number of different operations in a staggered manner.

According to this embodiment, 64-bit hardware may be used to process 128-bit data. A 128-bit register may be broken into four 32-bit elements, each of which represents a separate 32-bit value. At time T, the two ADD execution units perform ADDs first on the two lower 32-bit values, followed by an ADD on the higher 32-bit values at time T+1. In the case of a MUL operation, the MUL execution units behave in the same manner. This ability to use currently available 64-bit hardware to process 128-bit data represents a significant cost advantage to hardware manufacturers.

As described above, the ADD and MUL execution units according to the present embodiment are reused to reexecute a second ADD or MUL operation at a subsequent clock cycle. Of course, in the case of a partial-width packed data instruction, the execution units are reused but the operation is not necessarily reexecuted since power to the execution unit may be selectively shut down. At any rate, as described earlier, in order for this re-using or "staggered execution" to perform efficiently, this embodiment takes advantage of the statistical behavior of multimedia applications.

If a second ADD instruction follows a first ADD instruction, the second ADD may be delayed by a scheduling unit to allow the ADD execution units to complete the first ADD instruction, or more specifically on the higher order data segment of the first ADD instruction. The second ADD instruction may then begin executing. Alternatively, in an out-of-order processor, the scheduling unit may determine that a MUL instruction further down the instruction stream may be performed out-of-order. If so, the scheduling unit may inform the MUL execution units to begin processing the MUL instruction. If no MUL instructions are available for processing at time T+1, the scheduler will not issue an instruction following the first ADD instruction, thus allowing the ADD execution units time to complete the first ADD instruction before beginning the second ADD instruction.

Yet another embodiment of the invention allows for back-to-back ADD or MUL instructions to be issued by executing the instructions on the same execution units on half clock cycles instead of full clock cycles. Executing an

instruction on the half clock cycle effectively "double pumps" the hardware, i.e. makes the hardware twice as fast. In this manner, the ADD or MUL execution units may be available during each clock cycle to process a new instruction. Double pumped hardware would allow for the hardware units to execute twice as efficiently as single pumped hardware that executes only on the full clock cycle. Double pumped hardware requires significantly more hardware, however, to effectively process the instruction on the half clock cycle.

It will be appreciated that modifications and variations of the invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention. For example, although only two execution units are described above, any number of logic units may be provided.

Second Exemplary Embodiment Employing "Staggered Execution"

According to an alternate embodiment of the invention, the staggered execution of a full width operand is achieved by converting a full width macro instruction into at least two micro instructions that each operate on only half of the operands. As will be described further below, when the macro instruction specifies a partial-width packed data operation, better performance can be achieved by eliminating micro instructions that are not necessary for the determination of the partial-width result. In this manner, processor resource constraints are reduced and the processor is not unnecessarily occupied with inconsequential micro instructions. Although

the description below is written according to a particular register renaming method, it will be appreciated that other register renaming mechanisms may also be utilized consistent with the invention. The register renaming method as described below assumes the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement buffer, as described in detail in U.S. Patent No. 5,446,912. Alternate register renaming methods such as that described in U.S. Patent No. 5,197,132 may also be implemented.

Figure 9 conceptually illustrates one embodiment of a pipeline to perform operations on operands in a "staggered" manner by converting a macro instruction into a plurality of micro instructions that each processes a portion of the full width of the operands. It should be noted that various other stages of the pipeline, e.g. a prefetch stage, have not been shown in detail in order not to unnecessarily obscure the invention. As illustrated, at the decode stage of the pipeline, a full width macro instruction is received, specifying logical source registers, each storing a full width operand (e.g. 128-bit). By way of example, the described operands are 128-bit packed floating point data operands. In this example, the processor supports Y logical registers for storing packed floating point data. The macro instruction is converted into micro instructions, namely a "high order operation" and a "low order operation," that each cause the operation of the macro instruction to be performed on half the width of the operands (e.g., 64 bits).

The two half width micro instructions then move into a register renaming stage of the pipeline. The register renaming stage includes a

variety of register maps and reorder buffers. The logical source registers of each micro instruction are pointers to specific register entries in a register mapping table (e.g. a RAT). The entries in the register mapping table in turn point to the location of the physical source location in an ROB or in a retirement register. According to one embodiment, in order to accommodate the half width high and low order operations described above, a RAT for packed floating point data is provided with Y*2 entries. Thus, for example, instead of a RAT with the entries for 8 logical registers, a RAT is created with 16 entries, each addressed as "high" or "low." Each entry identifies a 64-bit source corresponding to either a high or a low part of the 128-bit logical register.

Each of the high and low order micro instructions thus has associated entries in the register mapping table corresponding to the respective operands. The micro instructions then move into a scheduling stage (for an out of order processor) or to an execution stage (for an in order processor). Each micro instruction retrieves and separately processes a 64-bit segment of the 128-bit operands. One of the operations (e.g. the lower order operation) is first executed by the 64-bit hardware units. Then, the same 64-bit hardware unit executes the higher order operation. It should be appreciated that zero or more instructions may be executed between the lower and higher order operations.

Although the above embodiment describes the macro instruction being divided into two micro instructions, alternate embodiments may divide the

macro instruction into more micro instruction. While Figure 9 shows that the packed floating point data is returned to a retirement register file with Y*2 64-bit registers, each designated as high or low, alternate embodiments may use a retirement register file with Y 128-bit registers. In addition, while one embodiment is described having a register renaming mechanism with a reorder buffer and retirement register files, alternate embodiments may use any register renaming mechanism. For example, the register renaming mechanism of U.S. Patent No. 5,197,132 uses a history queue and backup map.

Figure 10 is a timing chart that further illustrates the embodiment described in Figures 9. At time T, a macro instruction "ADD X, Y" enters the decode stage of the pipeline of Figure 9. By way of example, the macro instruction here is a 128-bit instruction. The 128-bit macro instruction is converted into two 64-bit micro instructions, namely the high order operation, "ADD X_H, Y_H" and the low order operation, "ADD X_L Y_L." Each micro instruction then processes a segment of data containing two data elements. For example, at time T, the low order operation may be executed by a 64-bit execution unit. Then at a different time (e.g., time T+N), the high order operation is executed by the same 64-bit execution unit. This embodiment of the invention is thus especially suitable for processing 128-bit instructions using existing 64-bit hardware systems without significant changes to the hardware. The existing systems are easily extended to include a new map to handle packed floating point, in addition to the existing logical register maps.

Referring now to Figure 11, decoding logic that may be employed according to one embodiment of the invention is described. Briefly, in the embodiment depicted, a plurality of decoders 1110, 1120, and 1130 each receive a macro instruction and convert it into a micro instruction. Then the micro operating are sent down the remainder of the pipeline. Of course, N micro instructions are not necessary for the execution of every macro instruction. Therefore, it is typically the case that only a subset of micro instructions are queued for processing by the remainder of the pipeline.

As described above, packed data operations may be implemented as two half width micro instructions (e.g., a high order operation and a low order operation). Rather than independently decoding the macro instruction by two decoders to produce the high and low order operations as would be typically required by prior processor implementations, as a feature of the present embodiment both micro instructions may be generated by the same decoder. In this example, this is accomplished by replication logic 1150 which replicates either the high or low order operation and subsequently modifies the resulting replicated operation appropriately to create the remaining operation. Importantly, as was described earlier, by carefully encoding the register address space, the registers referenced by the micro instructions (e.g., the logical source and destination registers) can be made to differ by a single bit. As a result, the modification logic 1160 in its most simple form may comprise one or more inverters to invert the appropriate bits to produce a high order operation from a low order operation and vice versa. In any

event, the replicated micro instruction is then passed to multiplexer 1170. The multiplexer 1170 also receives a micro instruction produced by decoder 1120. In this example, the multiplexer 1170, under the control of a validity decoder 1180, outputs the replicated micro instruction for packed data operations (including partial-width packed data operations) and outputs the micro instruction received from decoder 1120 for operations other than packed data operations. Therefore, it is advantageous to optimize the opcode map to simplify the detection of packed data operations by the replication logic 1150. For example, if only a small portion of the of the macro instruction needs to be examined to distinguish packed data operations from other operations, then less circuitry may be employed by the validity decoder 1180.

In an implementation that passes through source data elements to the logical destination register for purposes of executing partial-width packed data operations, in addition to selection logic similar to that described with respect to Figures 5A and 5C, logic may be included to eliminate ("kill") one of the high or low order operations. Preferably, for performance reasons, the extraneous micro instruction is eliminated early in the pipeline. This elimination may be accomplished according to the embodiment depicted by using a micro instruction selection signal output from micro instruction length determination circuitry 1190. The micro instruction length determination logic 1190 examines a portion of the macro instruction and produces the micro instruction selection signal which indicates a particular

combination of one or more micro instructions that are to proceed down the pipeline. In the case of a scalar SIMD instruction, only one of the resulting high and low order operations will be allowed to proceed. For example, the micro instruction selection signal may be represented as a bit mask that identifies those of the micro instructions that are to be retained and those that are to be eliminated. Alternatively, the micro instruction selection signal may simply indicate the number of micro instructions from a predetermined starting point that are to be eliminated or retained. Logic required to perform the elimination described above will vary depending upon the steering mechanism that guides the micro instructions through the remainder of the pipeline. For instance, if the micro instructions are queued, logic would may be added to manipulate the head and tail pointers of the micro instruction queue to cause invalid micro instructions to be overwritten by subsequently generated valid micro instructions. Numerous other elimination techniques will be apparent to those of ordinary skill in the art.

Although for simplicity only a single macro instruction is shown as being decoded at a time in the embodiment depicted, in alternative embodiments multiple macro instructions may be decoded concurrently. Also, it is appreciated that micro instruction replication has broader applicability than that illustrated by the above embodiment. For example, in a manner similar to that described above, full-width and partial-width packed data macro instructions may be decoded by the same decoder. If a prefix is used to distinguish full-width and partial width packed data macro

instructions, the decoder may simply ignore the prefix and decode both types of instructions in the same manner. Then, the appropriate bits in the resulting micro operations may be modified to selectively enable processing for either all or a subset of the data elements. In this manner, full-width packed data micro operations may be generated from partial-width packed data micro operations or vice versa, thereby reducing complexity of the decoder.

Thus, a method and apparatus for efficiently executing partial-width packed data instructions are disclosed. These specific arrangements and methods described herein are merely illustrative of the principles of the invention. Numerous modifications in form and detail may be made by those of ordinary skill in the art without departing from the scope of the invention. Although this invention has been shown in relation to a particular preferred embodiment, it should not be considered so limited. Rather, the invention is limited only by the scope of the appended claims.

CLAIMS

1. A processor comprising:
a plurality of registers;
a register renaming unit coupled to the plurality of registers to provide an architectural register file to store packed data operands, each of said packed data operands having a plurality of data elements;
a decoder, coupled to said register renaming unit, to decode a first and second set of instructions that each specify one or more registers in the architectural register file, each instruction in the first set of instructions specifying operations on all of the data elements stored in the specified one or more registers, each of the second set of instructions specifying an operation on only a subset of data element stored in a specified one or more registers; and
a partial-width execution unit, coupled to the decoder to execute operations specified by either of the first or the second set of instructions.

2. The processor of claim 1, wherein the subset of data elements stored in a specified one or more registers comprises corresponding least significant data elements.

3. The processor of claim 1, further comprising an execution unit to selectively perform a specified operation on one or more data elements

in the specified one or more registers depending upon which of the first or second set of instructions the specified operation is associated.

4. The procesor of claim 3, wherein the execution unit further comprises a plurality of multiplexers to select between a result of the specified operation and a predetermined value.

5. The processor of claim 3, wherein the execution unit further comprises a plurality of multiplexers to select between a data element of the one or more data elements and an identity function for input to the specified operation.

6. A method comprising the steps of:

receiving a single macro instruction specifying at least two logical registers in a packed data register file, wherein the two logical registers respectively store a first packed data operand and second packed data operand having corresponding data elements; and

independently operating on a first and second plurality of the corresponding data elements from said first and second packed data operands at different times using the same circuit to independently generate a first and second plurality of resulting data elements by

performing an operation specified by the single macro instruction on at least one pair of corresponding data elements in the first and second plurality corresponding

data elements to produce at least one resulting data element of the first and second plurality of resulting data elements, and

setting remaining resulting data elements of the first and second plurality of resulting data elements to one or more predetermined values; and

storing the first and second plurality of resulting data elements in a single logical register as a third packed data operand.

7. The method of claim 6, wherein the one or more predetermined values comprise values of data elements from either the first packed data operand or the second packed data operand.

8. The method of claim 6, wherein the one or more predetermined values comprise zero.

9. The method of claim 6, wherein the one or more predetermined values comprise a not-a-number (NaN) indication.

10. A processor substantially as herein described with reference to and as shown in each of the embodiments shown in Figures 2A-11 of the accompanying drawings.

11. A method substantially as herein described with reference to and as shown in each of the embodiments shown in Figures 2A-11 of the accompanying drawings.

The Patent Office

INVESTOR IN PEOPLE

Application No:	GB 9907221.7	**Examiner:**	Leslie Middleton
Claims searched:	1	**Date of search:**	28 October 1999

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.Q): G4A (AVL)

Int Cl (Ed.6): G06F 9/302

Other: Online: EPODOC, PAJ, WPI / EPOQUE

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
A	WO 9722924 A1 (Intel Corpn.) See Fig. 3A, and column 9	
A	WO 97/22923 A1 (Intel Corpn.) See Fig. 3A, and cols.6,9.	
A	WO 97/22921 A1 (Intel Corpn.) See Fig. 3A, & pp.15,21.	

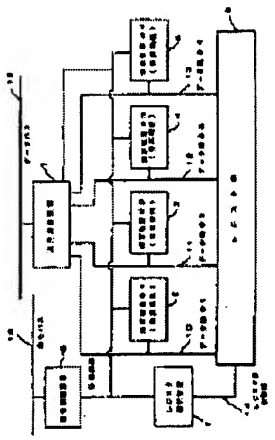
X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

MICROPROCESSOR, INFORMATION PROCESSOR AND GRAPHIC DISPLAY DEVICE USING IT

Publication number: JP3268024
Publication date: 1991-11-28
Inventor: FUKAYA MASAMICHI; KATSURA AKIHIRO; KOGA KAZUYOSHI; HOTTA TAKASHI
Applicant: HITACHI LTD
Classification:
- International: ***G06F7/53; G06F7/52; G06F9/305; G06F9/38; G06F15/80; G06T11/00;*** **G06F7/48; G06F9/305; G06F9/38; G06F15/76; G06T11/00;** (IPC1-7): G06F7/52; G06F15/72; G06F15/80
- European:
Application number: JP19900067064 19900319
Priority number(s): JP19900067064 19900319

Abstract of **JP3268024**

PURPOSE:To execute the picture element arithmetic operation at high speed by a small number of register resources by allocating plural arithmetic units having a multiplying function to one register unit, and actuating them simultaneously by an exclusive instruction. CONSTITUTION:Plural arithmetic units 2 - 5 having a multiplying function are provided, and also, one register designated by an instruction is divided into plural areas, and each arithmetic unit 2 - 5 is allowed to correspond to the respective areas. For instance, a register unit is divided into four areas, and the arithmetic units 2 - 5 are connected to each of them through data lines 10 - 13. Also, in an instruction controller 6, an instruction for actuating simultaneously the multiplying functions of the arithmetic units 2 - 5 is registered in advance, and in accordance with this instruction, a general arithmetic unit 1 executes such processings as an arithmetic operation and a data transfer, etc., to all areas of the register unit. In such a way, the information processing can be executed efficiently.



⑲日本国特許庁（JP）　⑪特許出願公開

⑫公開特許公報（A）　平3−268024

⑤1Int. Cl.⁵	識別記号	庁内整理番号	㊸公開　平成3年(1991)11月28日
G 06 F 7/52	310 A	2116−5B	
15/72	A	8125−5L	
15/80		7056−5L	

審査請求　未請求　請求項の数 5　（全7頁）

㊹発明の名称　マイクロプロセッサ、情報処理装置及びそれを用いた図形表示装置

㉑特　願　平2−67064
㉒出　願　平2(1990)3月19日

㉒発明者　深谷　正道　茨城県日立市久慈町4026番地　株式会社日立製作所日立研究所内
㉒発明者　桂　晃洋　茨城県日立市久慈町4026番地　株式会社日立製作所日立研究所内
㉒発明者　古賀　和義　茨城県日立市久慈町4026番地　株式会社日立製作所日立研究所内
㉒発明者　堀田　多加志　茨城県日立市久慈町4026番地　株式会社日立製作所日立研究所内
㉑出願人　株式会社日立製作所　東京都千代田区神田駿河台4丁目6番地
㉔代理人　弁理士　小川　勝男　外2名

明　細　書

1．発明の名称

マイクロプロセッサ，情報処理装置及びそれを用いた図形表示装置

2．特許請求の範囲

1．複数のレジスタからなるレジスタ群と、当該レジスタ群から一つのレジスタ単位を選択するレジスタ選択装置と、前記レジスタ内の異なるビット列の演算を行う少なくとも乗算機能を有する複数の演算装置と、前記レジスタ単位の全ビット長に対してデータ処理を行う汎用演算装置と、前記レジスタ群、汎用演算装置および演算装置を制御する制御装置とからなることを特徴とする情報処理装置。

2．請求項1記載の情報処理装置において、該演算装置の乗算結果のビット列の一部が、選択された該レジスタの分割された一部に格納されることを特徴とする情報処理装置。

3．請求項1又は2記載の情報処理装置において、前記演算装置間に演算補助信号を伝える信号線を設けたことを特徴とする情報処理装置。

4．請求項1乃至3記載の情報処理装置において、前記複数の演算装置を同時に起動する専用命令を備えたことを特徴とするマイクロプロセッサ。

5．請求項1乃至4記載の情報処理装置に、外部記憶装置および表示装置を設けたことを特徴とする図形表示装置。

3．発明の詳細な説明

〔産業上の利用分野〕

本発明は、電子技術を用いた情報処理装置に係り、特に高速に図形データなどを処理することを目的としたマイクロプロセッサおよび図形処理装置に適用できる。

〔従来の技術〕

従来の装置は、アイ・イー・イー・イー，コンピュータ　グラフィックス　アンド　アプリケーション、１９８９年７月号、第８５頁から第９４頁（IEEE Computer Graphics & Applications,July, 1987, pp85-94)に紹介されているマイクロプロセッサのように、単純な加算について一つのレジス

タを複数の領域に分割し、それぞれに対応する複数の加算器を用いてデータを処理する技術によって、並列画素演算などの情報処理を行なっていた。この装置は、グラフィックス処理における輝度補間などの単純な処理を行うのに適しているが、透明感を描出するアルファブレンド処理のように乗算を要する処理においては、多数の命令の組合せによって画素計算用の乗算プログラムを記述する必要があり、効果的な処理が困難であった。

〔発明が解決しようとする課題〕

本発明は、従来技術では効率的な処理が困難であった乗算を含む並列画素演算などの情報処理を効率的に実行できる手段を提供するものである。

〔課題を解決するための手段〕

上記目的を達成するために、本発明は、複数のレジスタからなるレジスタ群と、当該レジスタ群から一つのレジスタ単位を選択するレジスタ選択装置と、前記レジスタ内の異なるビット列の演算を行う少なくとも乗算機能を有する複数の演算装置と、前記レジスタ単位の全ビット長に対してデータ処理を行う汎用演算装置と、前記レジスタ群、汎用演算装置及び演算装置を制御する制御装置とから、情報処理装置を構成したものである。

〔作用〕

乗算機能を有する複数の演算装置を設け、さらに並列画素演算におけるレジスタ使用効率を向上させるために、命令で指定された一つのレジスタを複数の領域に分割し、それぞれの領域に前記演算装置を対応させたものである。演算結果の一部のビット列のみをレジスタに格納する手段を設けた場合には、レジスタの利用効率はさらに高まる。

また、桁上がり信号などの演算補助信号を伝える信号線を各演算装置間に設けた場合には、本発明で用いる演算装置を複数個組み合わせることによって、よりビット長の大きな演算装置として使用できる。これらのハードウエアは制御装置によって制御される。

本発明による情報処理装置はマイクロプロセッサの構成要素とすることができる。その場合にはマイクロプロセッサは専用の命令を備え、それを解釈する制御装置を内部に持つ。

以上の情報処理装置を陰極線管を用いた表示装置や印字装置に適用することにより、高性能な図形処理装置を実現できる。

〔実施例〕

第１図は本発明による情報処理装置の一実施例を示す構成図である。第１図において２，３，４，５は演算装置であり、それぞれ乗算機能を有する。汎用演算装置１は、データバス１５に接続され、命令制御装置６は、命令バス１６に接続されている。命令バスとデータバスは共用することもある。命令制御装置６は制御線９によって、命令に応じた制御信号を演算装置２，３，４，５、レジスタ選択装置７および汎用演算装置１に伝達し、それぞれの動作を制御する。命令制御装置６には演算装置２，３，４，５の乗算機能を同時に起動する命令が登録されている。制御線９は各装置に対し独立に与えられることもある。レジスタ選択装置７は、命令に従ってレジスタ群８の中から必要なレジスタ単位を選択する。１４はレジスタ群制御線である。汎用演算装置１が３２ビットのプロセッサの場合、レジスタ単位は原則として３２ビットであるが、倍精度演算などで用いるレジスタペアではレジスタ単位を６４ビットや１２８ビットとする場合もある。

本発明ではレジスタ単位を複数のビット列に分割して利用する。分割された各領域に対応して演算装置を設ける。第１図に示した実施例では、レジスタ単位を４領域に分割し、それぞれに演算装置２，３，４，５をデータ線１０，１１，１２，１３を介して接続する。レジスタ単位の分割数は、命令によって指定できる。演算装置の総数は、第１図に示した実施例では４であるが、レジスタ単位の分割数に応じて必要数だけ設ける。汎用演算装置１は、レジスタ単位の全領域に対して演算やデータ転送などの処理を行う。

このような構成のシステムを図形データの演算に用いると、極めて高速なグラフィックス処理を実現できる。以下、汎用演算装置１が３２ビットプロセッサのシステムにおいて、コンピュータグ

ラフィックスで多用されるアルファブレンド処理を実行した場合を例として説明する。一般に画素は、R（赤），G（緑），B（青）の3要素によって表現され、それぞれ例えば8ビットのデータを割り当てる。アルファブレンド処理は、二つの画素データ（R1，G1，B1），（R2，G2，B2）をP：Qの比で混ぜ合わせて透明感を描出する手法で、混合後の画素データ（R3，G3，B3）は次式で表される。

R3＝P×R1＋Q×R2

G3＝P×G1＋Q×G2

B3＝P×B1＋Q×B2

第6図は本発明に係る情報処理装置によって高速化できるアルファブレンドの模式図である。図中の小丸で囲まれた部分は画素を表す。この処理を32ビットの汎用演算装置によって行なうと6回の乗算と3回の加算を実行しなければならない。また、レジスタ単位である32ビット空間に8ビットデータを一つずつ割り当てると残りの24ビット分が無駄になってしまう欠点があった。本発明では、レジスタ単位の32ビット空間に8ビットのデータを最大4個一度に割当て、それぞれのデータを担当する乗算機能を有する演算装置2，3，4，5を設けて並列に処理することによって高速化およびレジスタ利用効率の向上を図った。また、複数の演算装置を同時に起動する専用の命令を備えることによって、効率の良いプログラミングが可能となる。従来技術の中には、8ビット単位の加算器を並列に動作させることによって加算の回数を減らすことを可能にしたものは存在したが、演算時間の大部分を占める乗算については何ら有効な手段を持っていなかった。従来技術では、加算の並列化のためのデータ構造を定義して32ビット空間に4個の8ビットデータを割当てたとしても、同じデータ構造に対する乗算手段を持っていないために、加算部分と乗算部分との間のデータ受渡しの際にデータ型変換処理を要するなどの欠点があった。しかし、本発明ではこれらの欠点は全て解決される。以上に述べた例は、32ビットの汎用演算装置と8ビットの演算装置を組み合わせた例であるが、これらのビット長はシステムの設計者によって任意に決められる。

第2図は、演算装置2，3，4，5が実行すべき演算内容の一例を示す図である。21は被乗数、22は乗数、23は演算結果である。被乗数21は32ビットの領域を8ビットずつ区切り、4個の整数データa，b，c，dが割り当てられている。乗数22には同様にe，f，g，hが割り当てられている。演算結果23も8ビットずつ区切られて、a×e、b×f、c×g、d×hのそれぞれ上位8ビットが与えられる。乗数22、被乗数21、演算結果23は、レジスタ群8の内部に格納される。また、乗数22において一つの8ビットデータhのみを指定し、演算結果23にa×h，b×h，c×h，d×hの上位8ビットを格納する命令を定義すれば、一つの乗数が4つの被乗数に共通な場合の演算効率を向上できる。

以上の例では、積の上位8ビットを演算結果としたが、下位ビットを演算結果としてレジスタに格納する場合もある。レジスタペアを用いれば64ビットの空間に4組の8ビット乗算で得られた16ビットの積4組を格納することもできる。乗数22、被乗数21のデータとしては、符号付き整数、符号なし整数、浮動小数点、固定小数点など、様々な型に適用可能である。また、ビット長も8ビット，16ビット，24ビット，32ビット，64ビット，128ビット，13ビットなどシステムに適した任意の値を選択することができる。命令制御装置6は、命令バス16から得られた情報に従ってデータの型、ビット長を判断する機能を持つ。

演算装置2，3，4，5は、命令制御装置6によって制御されるので乗算機能に加えて加減算機能や積和機能除算機能を併せて持つことが容易となる。

積和演算を実行するための演算装置2，3，4，5はそれぞれ加算器と乗算器が接続された構造となっている。第7図に積和演算を実行させるために演算装置2，3，4，5がとるべき構成の一実施例を示す。71は加算器、72は乗算器である。

以上に述べた演算内容に付随する情報を示すフラグは、汎用演算装置1の内部に持つ場合と、演算装置2，3，4，5にそれぞれ持つ場合と、レジスタ群8の内部に持つ場合と、外部に信号として出力する場合などがあり、システムによっては特にフラグを定義しないこともある。第8図は汎用レジスタにフラグを格納するフラグ方式の一実施例である。r1には加算の桁上がりや減算のボローを示すキャリーフラグCを、r2には符号を表すサインフラグSをr3には零を表現するZフラグを格納した例である。このようにフラグをレジスタ群8のレジスタ単位に格納する場合、レジスタ単位を演算結果を格納するレジスタと同型に複数の領域に分割し、対応する領域にそれぞれの演算に付随するフラグを格納する方式を用いれば、演算装置2，3，4，5で発生したフラグ信号を容易にレジスタ群8に格納することができる。

第3図は、本発明の他の実施例を示す構成図である。演算補助信号線30，31，32，33は、演算装置2，3，4，5に接続され、演算装置2，3，4，5のうち複数を組み合わせて有機的に利用する場合に用いる。例として、演算装置2，3，4，5を8ビットの加算器とし、演算補助信号を桁上がり信号として、演算補助信号線30，31，32を用いれば、演算装置2，3，4，5は全体として32ビットの加算器と等価になる。このとき演算補助信号線31を接続しなければ2組の16ビット加算器となる。また、演算装置2，3，4，5が8ビットのシフト演算器の場合には、演算補助信号線30，31，32，33を用いることによって32ビットのローテーション演算器となる。以上の機能の切替は命令制御装置6によって行われる。

第4図は、演算装置2，3，4，5のうち複数の乗算機能を同時に起動する命令の一実施例である。オペコード41は、演算内容に応じて定義される。また、オペランド42は演算に係るデータのソースおよびデスティネーションとなるレジスタ単位の指定を行う。r1，r2，r3はレジスタ単位を表す。この命令は、一つのレジスタ単位に対し複数の乗算器が同時に割り当てられる場合でも、1ステップで記述できることを特徴とし、本発明による情報処理装置がマイクロプロセッサの構成要素となるときに有効な手段となる。命令のオペコードやオペランドは、演算装置2，3，4，5の演算単位が8ビットの場合、16ビットの場合、演算装置が乗算だけを実行する場合、積和演算を実行する場合などに応じて表現が異なる複数個を定義することができる。また、命令を構成するビット列の一部を本発明に係る処理状態と、汎用演算装置1による処理状態を区別するために利用する。第9図は本発明に係る処理状態に切替えるためのビットを備えた命令の一実施例である。92は命令のビット列、91は処理状態切替用ビットである。処理状態切替用ビットを複数個備えれば、より複雑な制御が可能となる。

第5図は、本発明の他の実施例である。マイクロプロセッサ54は、本発明に係る情報処理装置の機能を含む。記憶装置51および表示装置52をバス53を介して本発明によるマイクロプロセッサ54に接続することによって、特に高速なアルファブレンド処理を実現する図形表示装置が得られる。フレームメモリ55は、画素データを記憶する。バス53に印字装置を接続すれば表示装置52に表示された図形を印字できる印字装置が得られる。

〔発明の効果〕

以上述べたように、本発明では複数の乗算機能を有する演算装置を一つのレジスタ単位に割当て、専用の命令によってそれらを同時に起動できるので、少ないレジスタ資源で高速な画素演算を実行するシステムを実現できる。特にグラフィックスにおけるアルファブレンド処理では、演算時間の大幅削減、プログラムの短縮によるメモリ利用効率の向上などの効果がある。

4．図面の簡単な説明

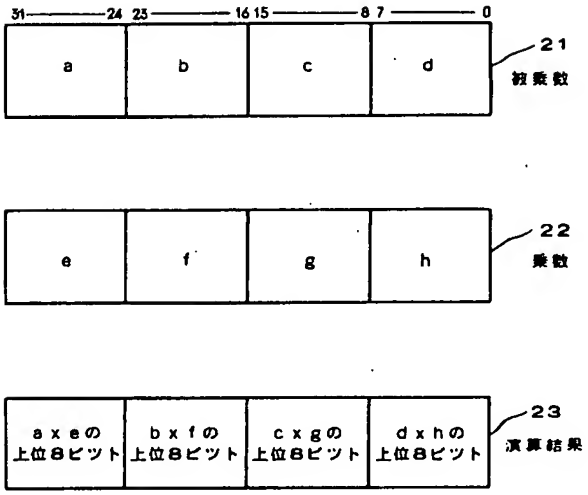
第1図は本発明の一実施例を示す構成図、第2図は本発明に係る演算装置が実行する演算内容の一例を示す図、第3図は本発明の他の実施例を示す構成図、第4図は本発明に係る命令の一実施例を

示す図、第５図は本発明のさらに他の実施例を示す構成図、第６図はアルファブレンドの模式図、第７図は積和用演算器の一実施例を示す構成図、第８図は本発明に係るフラグ方式の一実施例、第９図は処理状態を切替えるビットを備えた命令の一実施例を示す図である。

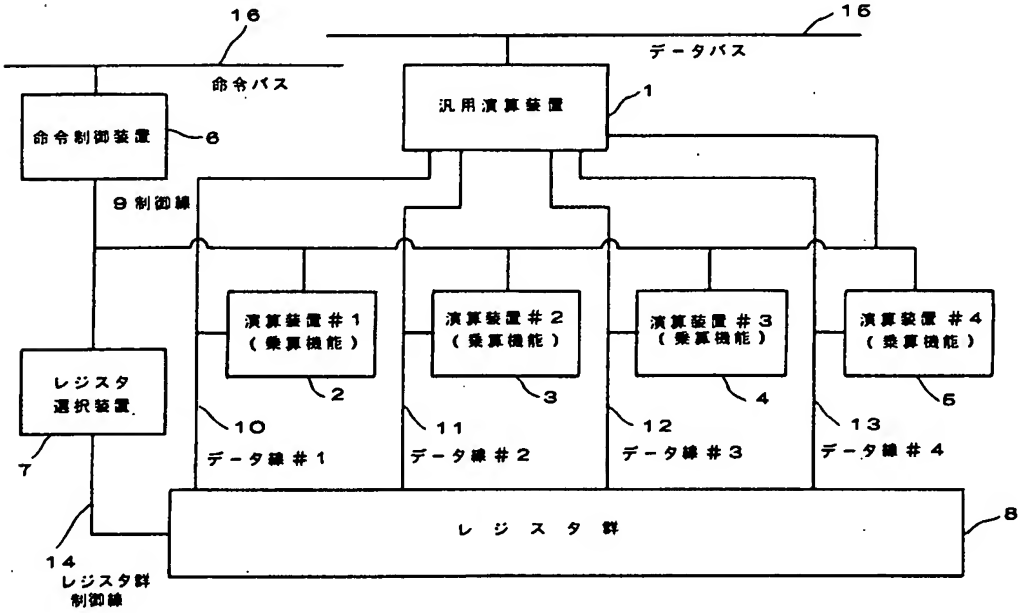
１…汎用演算装置、２，３，４，５…乗算機能を有する演算装置、６…命令制御装置、７…レジスタ選択装置、８…レジスタ群、９…制御線、１０，１１，１２，１３…データ線、１４…レジスタ群制御線、１５…データバス、１６…命令バス、２１…被乗数、２２…乗数、２３…演算結果、３０，３１，３２，３３…演算補助信号線、４１…オペコード、４２…オペランド、５１…記憶装置、５２…表示装置、５３…バス、５４…マイクロプロセッサ、５５…フレームメモリ、７１…加算器、７２…乗算器、９１…処理状態切替用ビット、９２…命令のビット列。

代理人　弁理士　小川勝男

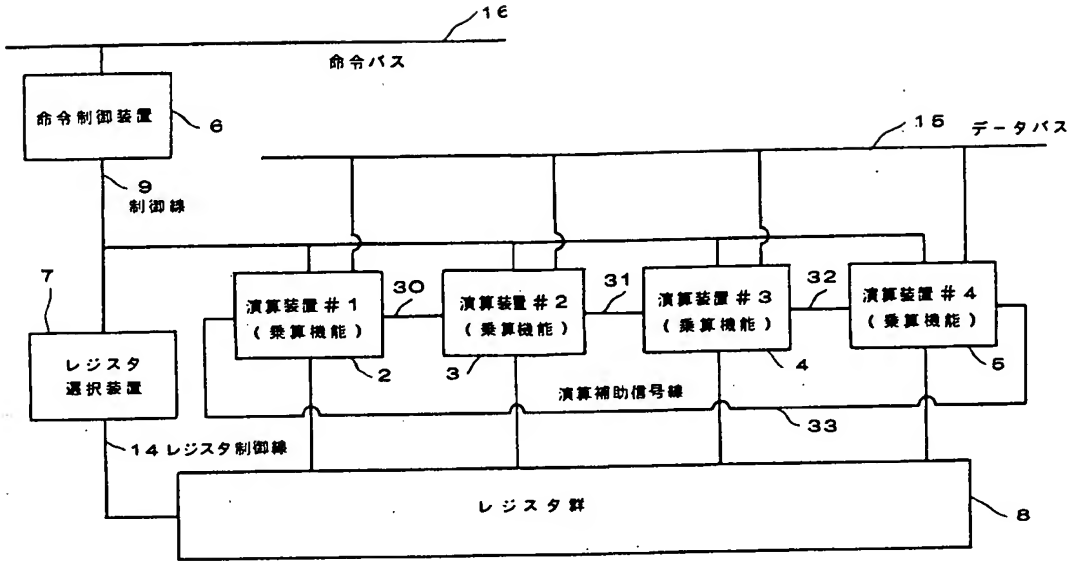
第　２　図



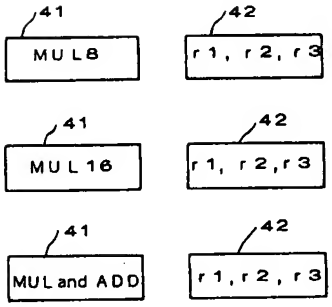
第　１　図



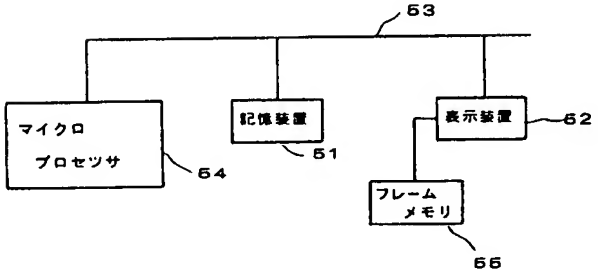
第 3 図



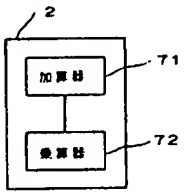
第 4 図



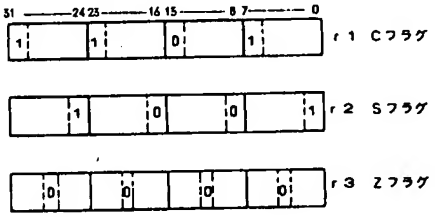
第 5 図



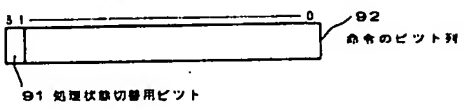
第 7 図



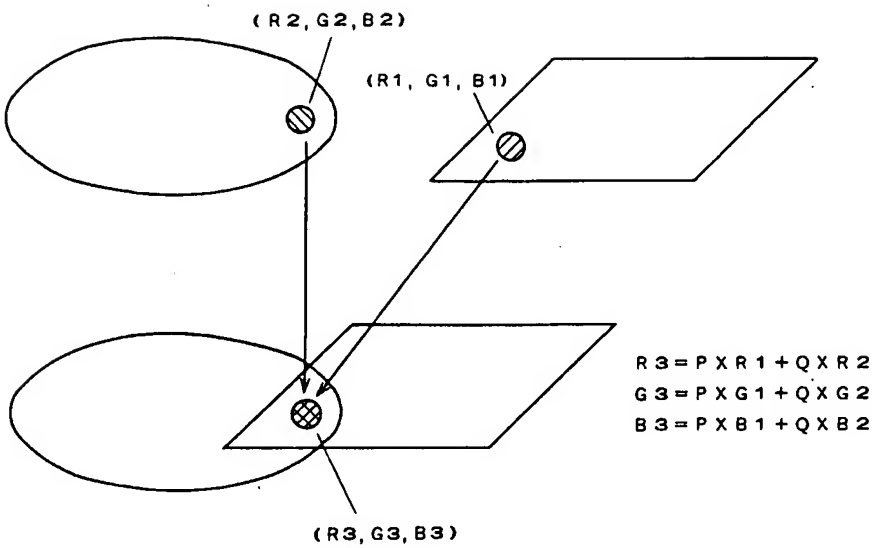
第 8 図



第 9 図



第 6 図



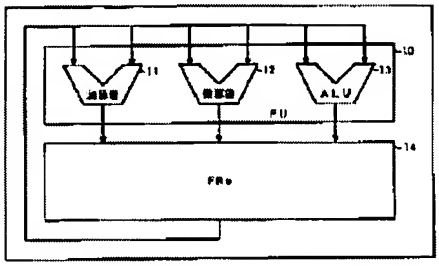
α ブレンド

MICROPROCESSOR

Publication number: JP6266554
Publication date: 1994-09-22
Inventor: SUZUKI KATSUNORI; FUJITA MAKOTO; KOGA KAZUYOSHI; FUJII HIDEKI
Applicant: HITACHI LTD
Classification:
- International: ***G06F7/00; G06F7/76; G06F9/30;*** **G06F7/00; G06F7/76; G06F9/30;** (IPC1-7): G06F9/30; G06F7/00; G06F15/72
- European:
Application number: JP19930056792 19930317
Priority number(s): JP19930056792 19930317

Abstract of **JP6266554**

PURPOSE:To provide a microprocessor suitable for a graphic system capable of remarkably accelerating a packing processing and an unpacking processing and to provide the graphic system to which the microprocessor is applied.
CONSTITUTION:This microprocessor for performing the geometrical processing of the graphic system is provided with a floating register, an adder 11, a multiplier 12 and an ALU 13 for performing a bit arithmetic processing inside a floating (floating point real number) arithmetic unit. Thus, since the microprocessor can perform the bit arithmetic processing with the floating register, the packing processing and the unpacking processing can be performed with the floating register at a high speed.



(19)日本国特許庁（ＪＰ）　(12) **公開特許公報(A)**　(11)特許出願公開番号

特開平6－266554

(43)公開日　平成６年(1994)９月22日

(51)Int.Cl.[5]		識別記号	庁内整理番号	F.I		技術表示箇所
Ｇ０６Ｆ	9/30	３５０　Ｅ	9189－5B			
	7/00					
	15/72					
			9188－5B	Ｇ０６Ｆ	7/00　１０１　Ｗ	

審査請求　未請求　請求項の数４　ＯＬ　（全 13 頁）

(21)出願番号　特願平5－56792

(22)出願日　平成５年(1993)３月17日

(71)出願人　000005108
株式会社日立製作所
東京都千代田区神田駿河台四丁目６番地

(72)発明者　鈴木　克徳
茨城県日立市大みか町七丁目１番１号　株式会社日立製作所日立研究所内

(72)発明者　藤田　良
茨城県日立市大みか町七丁目１番１号　株式会社日立製作所日立研究所内

(72)発明者　古賀　和義
茨城県日立市大みか町七丁目１番１号　株式会社日立製作所日立研究所内

(74)代理人　弁理士　小川　勝男

最終頁に続く

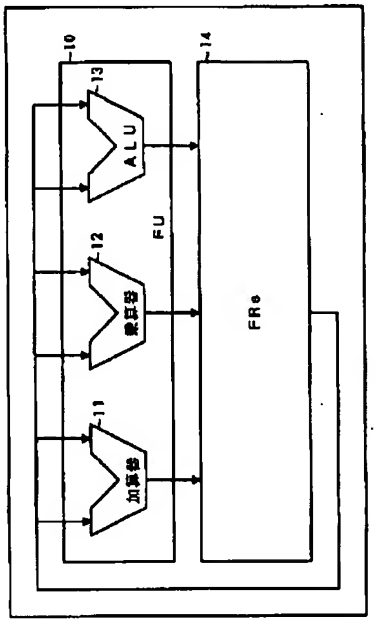
(54)【発明の名称】　マイクロプロセッサ

(57)【要約】

【目的】本発明は、pack処理，pnpack処理を著しく高速化することができるグラフイックス・システムに適したマイクロプロセッサ、及びそれを適用したグラフイックス・システムを提供することを目的としている。

【構成】前記目的を達成するために、本発明のグラフィックス・システムの幾何処理を行うマイクロプロセッサは、floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit 演算処理を行うＡＬＵを有する。これにより、本発明のマイクロプロセッサは、floatingレジスタ間でbit 演算処理を行うことができるので、pack処理，unpack処理をfloatingレジスタ間で高速に行うことが可能である。

図　1



1

【特許請求の範囲】

【請求項1】グラフィックス・システムの幾何処理を行うマイクロプロセッサにおいて、

floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit演算処理を行うＡＬＵを有し、（1）floatingレジスタにある1画素のデータを構成する各成分を示す複数のfloating型のデータを加算器によりfloating型からinteger(整数）型へ変換し、（2）前記変換結果である各成分を示す複数のinteger型のデータの有効なbit部分をＡＬＵにより1つの画素データに変換するpack処理をfloatingレジスタ間で行うことを特徴とするマイクロプロセッサ。

【請求項2】グラフィックス・システムの幾何処理を行うマイクロプロセッサにおいて、

floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit 演算処理を行うＡＬＵを有し、（1）floatingレジスタにある複数の成分から構成される1つの画素データをＡＬＵにより各成分を示す複数のinteger 型のデータに変換し、（2）前記変換結果である各成分を示す複数のinteger 型のデータを加算器によりinteger 型からfloating型へ変換するunpack処理をfloatingレジスタ間で行うことを特徴とするマイクロプロセッサ。

【請求項3】グラフィックス・システムの幾何処理を行うマイクロプロセッサにおいて、

floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit 演算処理を行うＡＬＵを有し、（1）floatingレジスタにある1画素のデータを構成する各成分を示す複数のfloating型のデータを加算器によりfloating型からinteger （整数）型へ変換し、（2）前記変換結果である各成分を示す複数のinteger 型のデータの有効なbit部分をＡＬＵにより1つの画素データに変換するpack処理をfloatingレジスタ間で行う命令を有することを特徴とするマイクロプロセッサ。

【請求項4】グラフィックス・システムの幾何処理を行うマイクロプロセッサにおいて、

floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit 演算処理を行うＡＬＵを有し、（1）floatingレジスタにある複数の成分から構成される1つの画素データをＡＬＵにより各成分を示す複数のinteger 型のデータに変換し、（2）前記変換結果である各成分を示す複数のinteger 型のデータを加算器によりinteger 型からfloating型へ変換するunpack処理をfloatingレジスタ間で行う命令を有することを特徴とするマイクロプロセッサ。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】本発明は、グラフィックス・システムに適したマイクロプロセッサ、及びそれを適用した

2

グラフィックス・システムに関する。更に詳細にいえば、グラフィックス・システムにおける画素データのpack処理とunpack処理、即ち、（1）1画素のデータを構成する各成分を示す複数のfloating型のデータをinteger 型へ変換し、前記変換結果である各成分を示す複数のデータの有効なbit 部分を1つの画素データに変換する処理と、（2）複数の成分から構成される1つの画素データを各成分を示す複数のデータに変換するbit 演算処理と、前記変換結果である各成分を示す複数のinteger 型のデータをfloating型へ変換する処理の高速化に関する。

【０００２】

【従来の技術】コンピュータ・グラフィックス・システムは、コンピュータの出力を図形として表示するものである。従来のグラフィックス・システムの図形表示方法は、「PEX Introduction and Overview (M.I.T., 1988, pp51－72)」に詳細が記載されている。

【０００３】グラフィックス・システムは、図形の基準点の座標変換と、図形，光源，視点の位置，色などの情報より図形がどの様に見えるか光源計算を行い図形の基準点の色を算出する幾何処理と、図形の基準点の情報からその図形の内部の画素を内挿補間により1画素ずつ展開して描画するレンダリング処理を行い、ディスプレイに表示する内容をビットマップ形式で保持する画像メモリに書き込む。グラフィックス・システムは、一般に、画素データを図形の座標データＸＹＺと色データＲＧＢ(Red, Green, Blue)で表現し、前記幾何処理ではfloating（浮動小数点実数）型で計算し、前記レンダリング処理ではinteger （整数）型で計算する。ここでは、説明を簡単にするために、画像メモリの画素データの色データＲＧＢの各成分が各8bit 、合計２４bit であるグラフィックス・システムについて説明する。

【０００４】前記の様に、floating型のｒ，ｇ，ｂをinteger 型のＩｒ，Ｉｇ，Ｉｂに変換し、各成分別々なＩｒ，Ｉｇ，Ｉｂの有効な各8bit を1つの２４bit のデータに変換して画像メモリの画素データを表現するため、グラフィックス・システムでは、ＲＧＢ各成分を示す複数のfloating型のデータｒ，ｇ，ｂを、ＲＧＢ各成分を示す複数のinteger 型のデータＩｒ，Ｉｇ，Ｉｂに変換するデータ型の変換処理と、前記変換結果であるＩｒ，Ｉｇ，Ｉｂの有効な各8bit 部分を1つの画素データに変換するbit 演算処理が画像メモリ書き込み時に必ず発生する。ここでは、この一連の処理をpack処理と呼ぶ。又、その逆変換処理、即ち、1つの画素データをＲＧＢ各成分を示す複数のinteger 型のデータＩｒ，Ｉｇ，Ｉｂに変換するbit 演算処理を行い、前記変換結果であるＩｒ，Ｉｇ，ＩｂをＲＧＢ各成分を示す複数のfloating型のデータｒ，ｇ，ｂに変換するデータ型の変換処理も発生する。ここでは、この一連の処理をunpack処理と呼ぶ。

3

【０００５】次に、従来のpack処理とunpack処理について説明する。

【０００６】まず、従来のpack処理について説明する。

【０００７】図４に、従来のpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

【０００８】但し、%grはinteger 演算を行うためのgeneral（汎用）レジスタ，%frはfloating演算を行うためのfloatingレジスタ，／＊…＊／はコメント（注釈）である。

【０００９】（１）初めに、Ｌ４０２，４０３，４０４では、初期状態として、ＲＧＢの各成分を示す複数のfloating型のデータｒ，ｇ，ｂがfloatingレジスタ１６，１７，１８番にある（４０２，４０３，４０４）。

【００１０】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

（２）そして、Ｌ４０６，４０７，４０８では、floating型のデータをinteger型のデータに変換する命令fcnvfxにより、前記データｒ，ｇ，ｂをinteger 型のデータＩｒ，Ｉｇ，Ｉｂへ変換するデータ型の変換処理を順次３回行う（４０６，４０７，４０８）。

【００１１】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

（３）次に、bit 演算処理を行うわけだが、一般にfloatingレジスタではbit 演算処理を行うことができない。そこで、bit 演算処理を行うために、Ｌ４１０，４１１，４１２では、floatingレジスタからgeneral レジスタへデータを転送する命令frtogrにより、前記データＩｒ，Ｉｇ，Ｉｂをfloatingレジスタ１６，１７，１８番からbit 演算処理が可能なgeneral レジスタ１６，１７，１８番へ転送するデータ転送処理を順次３回行う（４１０，４１１，４１２）。

（４）最後に、前記データＩｒ，Ｉｇ，Ｉｂの有効な各８bit を１つの画素データに変換するために、Ｌ４１４，４１５，４１６ではbit 演算処理を行う命令dep により、general レジスタ１６，１７，１８番にある前記Ｉｒ，Ｉｇ，Ｉｂの有効な各８bit をgeneral レジスタ１９番の各８bit の部分に設定するbit 演算処理を順次３回行う（４１４，４１５，４１６）。

【００１２】前記の様に従来のpack処理では、floatingレジスタでfloating型のデータｒ，ｇ，ｂをinteger 型のデータＩｒ，Ｉｇ，Ｉｂへ変換した後、floatingレジスタでは一般にbit 演算処理を行えないので、floatingレジスタからbit 演算処理が可能なgeneral レジスタへ転送し、general レジスタでbit 演算処理を行い１つの画素データに変換する。このため、floatingレジスタからgeneral レジスタへのデータ転送処理と、更に、順次３回のbit 演算処理が必要となり、pack処理に多大な時間を要するという問題を有する。

【００１３】次に、従来のunpack処理について説明する。

4

【００１４】図５に、従来のunpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

【００１５】（１）初めに、Ｌ５０２では、初期状態として、ＲＧＢの各成分Ｉｒ，Ｉｇ，Ｉｂの各８bit を既にpack処理した画素データがgeneral レジスタ１９番にある（５０２）。

【００１６】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

（２）そして、Ｌ５０４，５０５，５０６ではbit 演算処理を行う命令extru により、前記Ｉｒ，Ｉｇ，Ｉｂ各８bit をそれぞれgeneral レジスタ16，17，18番に設定するbit 演算処理を順次３回行う（５０４，５０５，５０６）。

（３）次に、データ型の変換処理を行うわけだが、一般にgeneral レジスタではデータ型の変換処理を行うことができない。そこで、データ型の変換処理を行うために、Ｌ５０８，５０９，５１０ではgeneral レジスタからfloatingレジスタへデータを転送する命令grtofrにより、前記データＩｒ，Ｉｇ，Ｉｂをgeneral レジスタ１６，１７，１８番からデータ型の変換処理が可能なfloatingレジスタ１６，１７，１８番へ転送するデータ転送処理を順次３回行う（５０８，５０９，５１０）。

【００１７】（４）最後に、Ｌ５１２，５１３，５１４ではinteger 型のデータをfloating型のデータに変換する命令fcnvxfにより、floatingレジスタ１６，１７，１８番にある前記データＩｒ，Ｉｇ，Ｉｂをfloating型のデータｒ，ｇ，ｂへ変換するデータ型の変換処理を順次３回行う（５１２，５１３，５１４）。

【００１８】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

前記の様に従来のunpack処理では、１つの画素データをgeneralレジスタでbit演算処理した後、general レジスタからデータ型の変換処理が可能なfloatingレジスタへ転送し、floatingレジスタでinteger 型のデータをfloating型のデータへ変換する。このため、general レジスタからfloatingレジスタへのデータ転送処理と、更に、順次３回のbit 演算処理を必要となり、unpack処理に多大な時間を要するという問題を有する。

【００１９】

【発明が解決しようとする課題】前記の様に従来のpack処理では、floatingレジスタからgeneral レジスタへのデータ転送処理と、更に、順次３回のbit 演算処理を必要とし、pack処理に多大な時間を要するという問題を有する。

【００２０】又、従来のunpack処理では、general レジスタからfloatingレジスタへのデータ転送処理と、更に、順次３回のbit 演算処理を必要とし、unpack処理に多大な時間を要するという問題を有する。

【００２１】従って、本発明は、前記問題点を解決して、pack処理，unpack処理を著しく高速化することがで

5

きるグラフィックス・システムに適したマイクロプロセッサ、及びそれを適用したグラフィックス・システムを提供することを目的としている。

【0022】

【課題を解決するための手段】前記目的を達成するために、本発明のグラフィックス・システムの幾何処理を行うマイクロプロセッサは、floating（浮動小数点実数）演算ユニット内に、floatingレジスタと、加算器と、乗算器と、bit 演算処理を行うＡＬＵを有する。これにより、本発明のマイクロプロセッサは、floatingレジスタ間でbit 演算処理を行うことができるので、floatingレジスタ間でpack処理，unpack処理を高速に行うことが可能となる。

【0023】以上の様に構成すれば、グラフィックス・システムにおける画素データのpack処理，unpack処理を高速に行うことができるグラフィックス・システムに適したマイクロプロセッサを構成できる。又、それをグラフィックス・システムに適用することにより、画素データのpack処理，unpack処理を著しく高速に行うことができるグラフィックス・システムを構成できる。

【0024】

【作用】本発明のグラフィックス・システムの幾何処理を行うマイクロプロセッサは、pack処理において、floatingレジスタにあるfloating型のデータを加算器によりinteger 型のデータへ変換し、前記変換結果であるinteger 型のデータの有効なbit 部分をＡＬＵによりそのままfloatingレジスタでbit 演算処理を行い１つの画素データに変換する。この様に、本発明のマイクロプロセッサは、floatingレジスタからgeneral レジスタへのデータ転送処理を行わないので、pack処理を著しく高速化することが可能となる。

【0025】又、本発明のグラフィックス・システムの幾何処理を行うマイクロプロセッサは、unpack処理において、floatingレジスタにある複数の成分から構成される１つの画素データをＡＬＵによりそのままfloatingレジスタで各成分を示す複数のinteger 型のデータに変換するbit演算処理を行い、前記変換結果であるinteger型のデータを加算器によりfloating型のデータへ変換する。この様に、本発明のマイクロプロセッサは、general レジスタからfloatingレジスタへのデータ転送処理を行わないので、unpack処理を著しく高速化することが可能となる。

【0026】以上の様に、本発明のマイクロプロセッサは、グラフィックス・システムにおける画素データのpack処理，unpack処理を著しく高速化することができる。又、それを適用したグラフィックス・システムは、グラフィックス・システムにおける画素データのpack処理，unpack処理を著しく高速化することが可能となる。

【0027】

【実施例】以下、実施例を図面によって詳細に説明する。

6

【0028】図１０は、本発明のグラフィックス・システムの一実施例を示す構成図である。

【0029】本発明のグラフィックス・システムは、幾何処理を行うマイクロプロセッサＣＰＵ（１０１），メモリ（１０３）等を制御するコントローラ（１０２），レンダリング処理を行う描画プロセッサ（１０４），描画した画像を保持する画像メモリ（１０５），画像を表示するディスプレイ（１０６）から構成される。

【0030】ＣＰＵ（１０１）は、アプリケーションを実行し幾何処理を行い図形の基準点の画素データとグラフィックス・コマンド（描画コマンド）を発行し、コントローラ（１０２）を通して描画プロセッサ（１０４）に送出する。描画プロセッサ（１０４）は、図形の基準点の画素データとグラフィックス・コマンドから図形の内部の画素を内挿補間により１画素ずつ展開して描画するレンダリング処理を行い、ディスプレイに表示する内容をビットマップ形式で保持する画像メモリ（１０５）に書き込み、画像をディスプレイ（１０６）に表示する。

【0031】グラフィックス・システムは、画素データを、幾何処理ではfloating（浮動小数点実数）型で計算し、レンダリング処理ではinteger （整数）型で計算する。ここでは、説明を簡単にするために、画像メモリの画素データの色データＲＧＢの各成分が各８bit，合計２４bit であるグラフィックス・システムについて説明する。

【0032】まず、幾何処理を行うマイクロプロセッサＣＰＵ（１０１）について説明する。

【0033】図２は、本発明のマイクロプロセッサの一実施例を示す構成図である。

【0034】本発明のマイクロプロセッサは、マイクロプロセッサと外部を接続するSystem Interface（２１），命令キャッシュＩＣ（２２），データキャッシュＤＣ（２３），マイクロプロセッサの各ユニットを制御するControler（２４），integer 演算を行うためのgeneral レジスタＧＲｓ（２５），integer 演算を実際に行うinteger 演算ユニットＩＵ（２６），floating演算を行うためのfloatingレジスタＦＲｓ（２７），floating演算を実際に行うfloating演算ユニットＦＵ（２８）から構成される。

【0035】本発明のマイクロプロセッサは、System Interface（２１）を介して、命令とデータを読み込み、命令キャッシュＩＣ（２２），データキャッシュＤＣ（２３）に格納し、命令に従ってControler がマイクロプロセッサの各ユニットを制御し、必要なデータがＧＲｓ（２５），ＦＲｓ（２７）に格納され、ＩＵ（２６），ＦＵ（２８）が命令の処理を実行する。

【0036】次に、ＦＵ（２８）の構成について詳細に説明する。

7

【００３７】図１は、本発明のマイクロプロセッサのＦＵとＦＲｓの一実施例を示す構成図である。

【００３８】ＦＵ（１０）は、加減算やデータ型の変換などの演算を行う加算器（１１），乗除算や平方根などの演算を行う乗算器（１２），pack処理，unpack処理で必要なbit 演算処理を行うＡＬＵ（１３）から構成される。ＦＵ（１０）は、命令に従って、ＦＲｓ（１４）のデータを読み込み、加算器（１１），乗算器（１２），ＡＬＵ（１３）で演算を行い、その結果のデータをＦＲｓ（１４）に書き込む。次に、本発明のマイクロプロセッサにおける高速なpack処理とunpack処理について詳細に説明する。

【００３９】まず、pack処理について説明する。

【００４０】図６に、本発明のマイクロプロセッサにおける高速なpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

【００４１】（１）初めに、Ｌ６０２，６０３，６０４では、初期状態として、ＲＧＢの各成分を示す複数のfloating型のデータｒ，ｇ，ｂがfloatingレジスタ１６，１７，１８番にある（６０２，６０３，６０４）。

【００４２】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

（２）次に、Ｌ６０７では、floating型のデータをinteger 型のデータへ変換してbit 演算処理を行う命令fcnvfxdep により、前記floatingレジスタ１６番にあるデータｒをinteger 型のデータＩｒへ変換するデータ型の変換処理を前記加算器（１１）で行い、更に、前記データＩｒの有効な８bit をfloatingレジスタ１９番の所定の８bit の部分に設定するbit 演算処理を前記ＡＬＵ（１３）で行う（６０７）。同様に、Ｌ６０８，６０９では、前記データｇ，ｂについて同様な処理を行う（６０８，６０９）。この様にして、ｆｌｏａｔｉｎｇ型のデータｒ，ｇ，ｂをinteger 型のデータＩｒ，Ｉｇ，Ｉｂに変換し、integer 型のデータＩｒ，Ｉｇ，Ｉｂの有効な各８bit を１つの画素データに変換する。

【００４３】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

前記の様に本発明のマイクロプロセッサにおける高速なpack処理では、floatingレジスタでfloating型のデータをinteger 型のデータへ変換し、従来のpack処理では必要としていたfloatingレジスタからgeneral レジスタへのデータ転送処理を行わずに、そのままfloatingレジスタでbit 演算処理を行い画素データに変換する。これにより、floatingレジスタからgeneral レジスタへのデータ転送処理が不要となり、pack処理を著しく高速化することができる。

【００４４】次に、unpack処理について説明する。

【００４５】図７に、本発明のマイクロプロセッサにおける高速なunpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

8

【００４６】（１）初めに、Ｌ７０２では、初期状態として、ＲＧＢの各成分Ｉｒ，Ｉｇ，Ｉｂ各８bit を既にpack処理した画素データがfloatingレジスタ１９番にある（７０２）。

【００４７】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

（２）次に、Ｌ７０５では、bit演算処理を行ってinteger型のデータをfloating型のデータへ変換する命令fextrucnvxfにより、前記floating レジスタ１９番にある画素データのデータＩｒの８bit をfloatingレジスタ１６番に設定するbit 演算処理を前記ＡＬＵ（１３）で行い、更に、前記データＩｒをfloating型のデータｒへ変換するデータ型の変換処理を前記加算器（１１）で行う（７０５）。同様に、Ｌ７０６，７０７では、前記データＩｇ，Ｉｂについて同様な処理を行う（７０６，７０７）。この様にして、前記画素データの各成分Ｉｒ，Ｉｇ，Ｉｂの各８bit を、各成分を示す複数のfloating型のデータｒ，ｇ，ｂへ変換する。

【００４８】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

前記の様に本発明のマイクロプロセッサにおける高速なunpack処理では、従来のunpack処理では必要としていたgeneral レジスタからfloatingレジスタへのデータ転送処理を行わずに、そのままfloatingレジスタで画素データのbit 演算処理を行い、integer 型のデータをfloating型のデータへ変換する。これにより、general レジスタからfloatingレジスタへのデータ転送処理が不要となり、unpack処理を著しく高速化することができる。

【００４９】次に、別のＦＵ（２８）の構成について詳細に説明する。

【００５０】図１に示した本発明のマイクロプロセッサのＦＵ（１０）は、加算器（１１），乗算器（１２），ＡＬＵ（１３）を１組ずつ有しているが、図３に示したように複数組有しても良い。

【００５１】図３は、加算器，乗算器，ＡＬＵを３組有する本発明のマイクロプロセッサのＦＵとＦＲｓの一実施例を示す構成図である。

【００５２】ＦＵ（３１０）は、加減算やデータ型の変換などの演算を行う加算器（３０１，３０２，３０３），乗除算や平方根などの演算を行う乗算器（３０４，３０５，３０６），pack処理，unpack処理で必要なbit 演算処理を行うＡＬＵ（３０７，３０８，３０９）から構成される。ＦＵ（３１０）は、命令に従って、ＦＲｓ（３１１）のデータを読み込み、加算器（３０１，３０２，３０３），乗算器（３０４，３０５，３０６），ＡＬＵ（３０７，３０８，３０９）で演算を行い、その結果のデータをＦＲｓ（３１１）に書き込む。

【００５３】次に、本発明のマイクロプロセッサの別のＦＵの構成における高速なpack処理とunpack処理について詳細に説明する。

9

【００５４】図８に、本発明のマイクロプロセッサにおける高速なpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

【００５５】（１）初めに、Ｌ８０２，８０３，８０４では、初期状態として、ＲＧＢの各成分を示す複数のfloating型のデータｒ，ｇ，ｂがfloatingレジスタ１６，１７，１８番にある（８０２，８０３，８０４）。

【００５６】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

（２）次に、Ｌ８０７では、３つのfloating型のデータをinteger型のデータへ変換してbit 演算処理を行う命令fcnvfxpack３により、前記floatingレジスタ１６，１７，１８番にあるデータｒ，ｇ，ｂをinteger 型のデータＩｒ，Ｉｇ，Ｉｂへ変換するデータ型の変換処理を前記加算器（３０１，３０２，３０３）で行い、更に、前記データＩｒ，Ｉｇ，Ｉｂの有効な各８bit をfloatingレジスタ１９番の所定の８bit の部分に設定するbit 演算処理を前記ＡＬＵ（307,３０８，３０９）で行う（６０７）。この様にして、floating型のデータｒ，ｇ，ｂをinteger 型のデータＩｒ，Ｉｇ，Ｉｂに変換し、integer 型のデータＩｒ，Ｉｇ，Ｉｂの有効な各８bit を１つの画素データに変換する。

【００５７】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

前記の様に本発明のマイクロプロセッサにおける高速なpack処理では、floatingレジスタでfloating型のデータをinteger 型のデータへ変換し、従来のpack処理では必要としていたfloatingレジスタからgeneral レジスタへのデータ転送処理を行わずに、更に、前記実施例ではそのままfloatingレジスタで順次３回必要としていたbit 演算処理を１回のbit 演算処理で行い画素データに変換する。これにより、floatingレジスタからgeneral レジスタへのデータ転送処理が不要となり、更に、bit 演算処理を１回に削減しているので、pack処理を著しく高速化することができる。

【００５８】図９に、本発明のマイクロプロセッサにおける高速なunpack処理のアセンブリ言語によるプログラムとレジスタの状態の例を示す。

【００５９】（１）初めに、Ｌ９０２では、初期状態として、ＲＧＢの各成分Ｉｒ，Ｉｇ，Ｉｂ各８bit を既にpack処理した画素データがfloatingレジスタ１９番にある（９０２）。

【００６０】（Ｉｒ，Ｉｇ，Ｉｂはinteger 型，０≦Ｉｒ，Ｉｇ，Ｉｂ≦２５５）

（２）次に、Ｌ９０５では、３回分のbit 演算処理を行ってfloating型のデータをinteger 型のデータへ変換する命令funpack３cnvxfにより、前記floatingレジスタ１９番にある画素データのデータＩｒ，Ｉｇ，Ｉｂの各８bit をfloatingレジスタ１６，１７，１８番に設定するbit 演算処理を前記ＡＬＵ（３０７，３０８，３０９）

10

で行い、更に、前記データＩｒ，Ｉｇ，Ｉｂをfloating型のデータｒ，ｇ，ｂへ変換するデータ型の変換処理を前記加算器（３０１，３０２，３０３）で行う（９０６，９０７，９０８）。この様にして、前記画素データの各成分Ｉｒ，Ｉｇ，Ｉｂの各８bit を、各成分を示す複数のfloating型のデータｒ，ｇ，ｂへ変換する。

【００６１】（ｒ，ｇ，ｂはfloating型，０.０≦ｒ，ｇ，ｂ≦２５５.０）

前記の様に本発明のマイクロプロセッサにおける高速なunpack処理では、従来のunpack処理では必要としていたgeneral レジスタからfloatingレジスタへのデータ転送処理を行わずに、更に、前記実施例ではそのままfloatingレジスタで順次３回必要としていたbit 演算処理を１回のbit 演算処理で画素データのbit 演算処理を行い、integer 型のデータをfloating型のデータへ変換する。これにより、general レジスタからfloatingレジスタへのデータ転送処理が不要となり、更に、bit 演算処理を１回に削減しているので、unpack処理を著しく高速化することができる。

【００６２】以上、詳細に説明した様に、本発明のマイクロプロセッサは、グラフィックス・システムにおける画素データのpack処理，unpack処理において、floatingレジスタとgeneral レジスタ間のデータ転送処理を不要にし、更に、３回分のbit 演算処理を１回のbit 演算処理に削減している。これにより、本発明のマイクロプロセッサは、pack処理，unpack処理を著しく高速化することができる。

【００６３】又、本発明のマイクロプロセッサをＣＰＵに用いたグラフィックス・システムも、グラフィックス・システムにおける画素データのpack処理，unpack処理を著しく高速化することができる。

【００６４】

【発明の効果】以上、詳細に説明した様に、本発明のグラフィックス・システムに適したマイクロプロセッサ、及びそれを適用したグラフィックス・システムによれば、グラフィックス・システムにおける画素データに関するpack処理，unpack処理を著しく高速化することができるという特有の効果を奏する。

【図面の簡単な説明】

【図１】本発明のマイクロプロセッサのＦＵとＦＲｓの一実施例を示す構成図である。

【図２】本発明のマイクロプロセッサの一実施例を示す構成図である。

【図３】本発明のマイクロプロセッサの一実施例を示す構成図である。

【図４】従来のpack処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図５】従来のunpack処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図６】本発明のマイクロプロセッサにおける高速なpa

11　　12

ck処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図7】本発明のマイクロプロセッサにおける高速なunpack処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図8】本発明のマイクロプロセッサにおける高速なpack処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図9】本発明のマイクロプロセッサにおける高速なunpack処理のアセンブリ言語によるプログラムとレジスタの状態を示す図である。

【図10】本発明のグラフィックス・システムの一実施例を示す構成図である。

【符号の説明】

10…FU、11…加算器、12…乗算器、13…ALU、14，27，311…FRs、21…System Interface、22…IC、23…DC、24…Controler 、25…GRs、26…IU、28，310…FU、301，302，303…加算器、304，305，306…乗算器、307，308，309…ALU。

【図1】

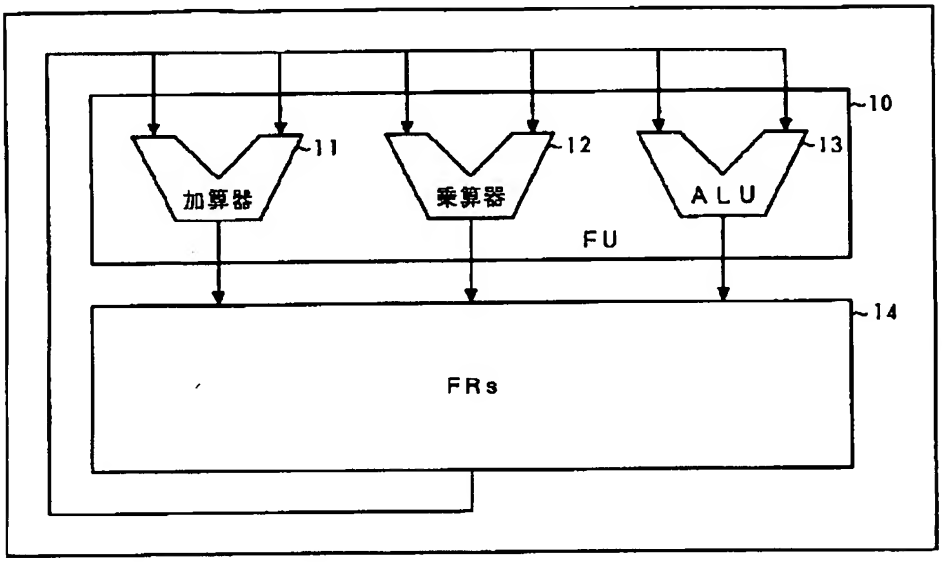
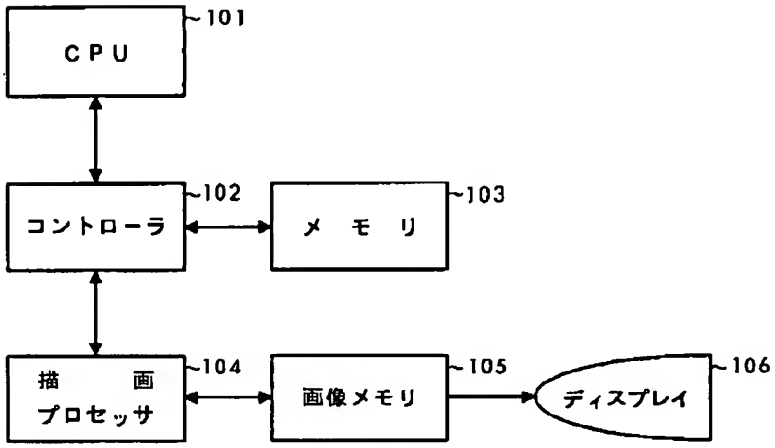


図1

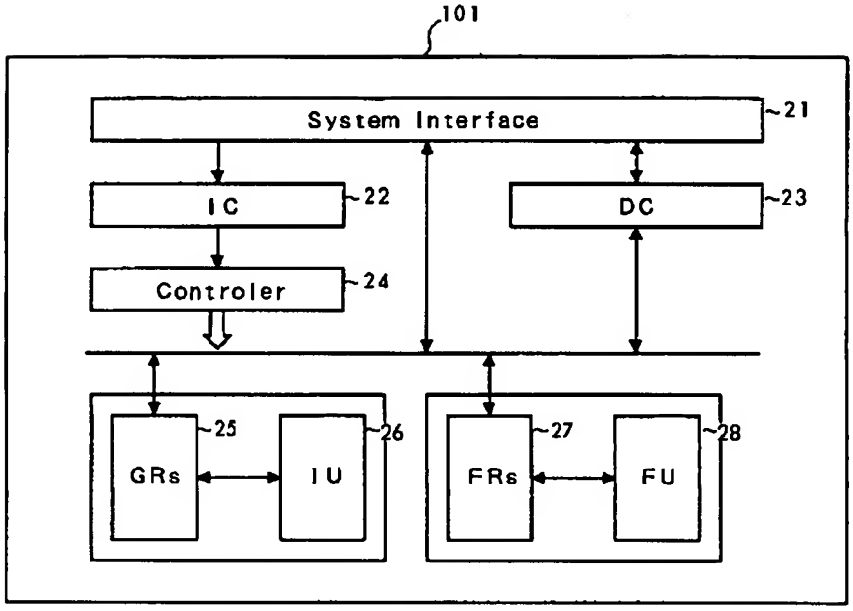
【図10】

図 10



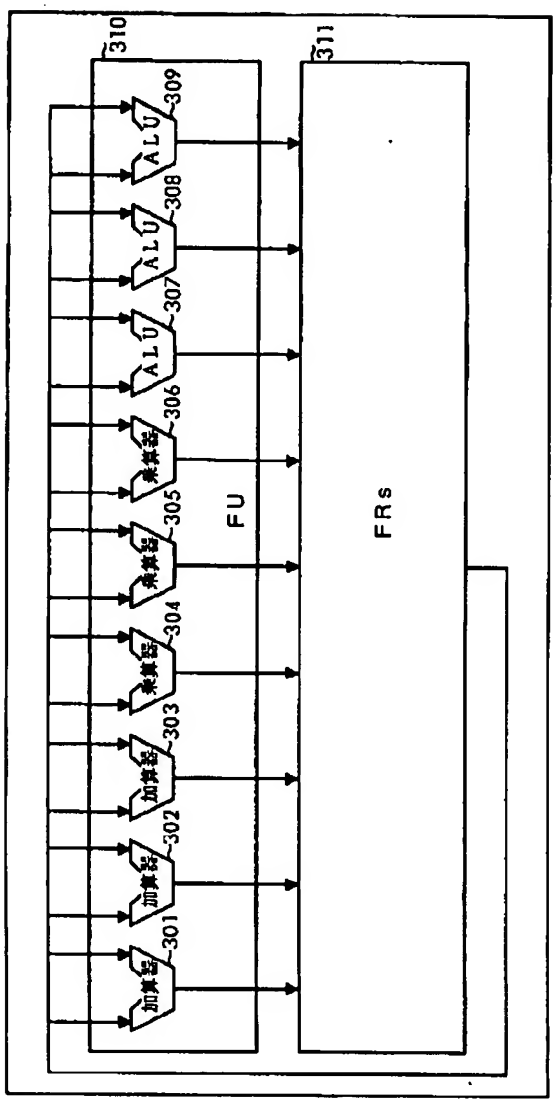
【図2】

図　2


101
System Interface
21
IC
22
DC
23
Controler
24
GRs
25
IU
26
FRs
27
FU
28

【図3】

図　3



【図7】

図　7

行番号	プログラム	レジスタの状態
L701	/* 初期状態：%fr19はIr,Ig,Ibの各8bitを	8bit 8bit 8bit 8bit
L702	packした画素データ */	%fr19 = Ir Ig Ib ～702
L703	/* %fr19をunpackし、	
L704	integer型からfloating型へ変換 */	32bit
L705	fextrucnvxf %fr19, 7,8,%fr16 /* %fr16 = r */	%fr16 = r ～705
L706	fextrucnvxf %fr19,15,8,%fr17 /* %fr17 = g */	%fr17 = g ～706
L707	fextrucnvxf %fr19,23,8,%fr18 /* %fr18 = b */	%fr18 = b ～707

【図4】

図 4

行番号	プログラム	レジスタの状態
L401	/* 初期状態 */	32bit
L402	/* %fr16 = r */	%fr16 = r ～402
L403	/* %fr17 = g */	%fr17 = g ～403
L404	/* %fr18 = b */	%fr18 = b ～404
L405	/* floating型をinteger型へ変換 */	8bit 8bit 8bit 8bit
L406	fcnvfx %fr16,%fr16 /* %fr16 = Ir */	%fr16 = Ir ～406
L407	fcnvfx %fr17,%fr17 /* %fr17 = Ig */	%fr17 = Ig ～407
L408	fcnvfx %fr18,%fr18 /* %fr18 = Ib */	%fr18 = Ib ～408
L409	/* floatingﾚｼﾞｽﾀからgeneralﾚｼﾞｽﾀへ転送 */	
L410	frtogr %fr16,%gr16 /* %gr16 = Ir */	%gr16 = Ir ～410
L411	frtogr %fr17,%gr17 /* %gr17 = Ig */	%gr17 = Ig ～411
L412	frtogr %fr18,%gr18 /* %gr18 = Ib */	%gr18 = Ib ～412
L413	/* Ir,Ig,Ib各8bitを%gr19にpack処理する */	
L414	dep %gr16, 7,8,%gr19	%gr19 = Ir ～414
L415	dep %gr17,15,8,%gr19	%gr19 = Ir Ig ～415
L416	dep %gr18,23,8,%gr19	%gr19 = Ir Ig Ib ～416

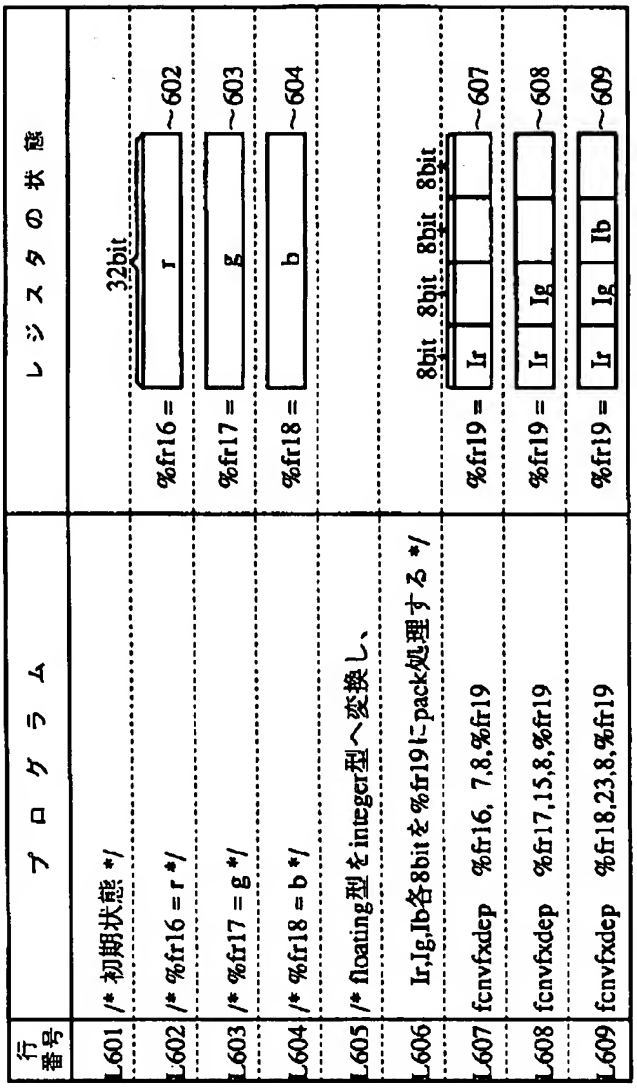
【図5】

図　5

行番号	プログラム	レジスタの状態
L501	/* 初期状態：%gr19はlr,lg,lbの各8bitを	8bit 8bit 8bit 8bit
L502	packした画素データ */	%gr19 = [lr][lg][lb][ ] ~502
L503	/* %gr19をunpack処理する */	
L504	extru %gr19, 7,8,%gr16 /* %gr16 = lr */	%gr16 = [ ][ ][ ][lr] ~504
L505	extru %gr19,15,8,%gr17 /* %gr17 = lg */	%gr17 = [ ][ ][ ][lg] ~505
L506	extru %gr19,23,8,%gr18 /* %gr18 = lb */	%gr18 = [ ][ ][ ][lb] ~506
L507	/* generalﾚｼﾞｽﾀからfloatingﾚｼﾞｽﾀへ転送 */	
L508	grtofr %gr16,%fr16 /* %fr16 = lr */	%fr16 = [ ][ ][ ][lr] ~508
L509	grtofr %gr17,%fr17 /* %fr17 = lg */	%fr17 = [ ][ ][ ][lg] ~509
L510	grtofr %gr18,%fr18 /* %fr18 = lb */	%fr18 = [ ][ ][ ][lb] ~510
L511	/* integer型をfloating型へ変換 */	32bit
L512	fcnvxf %fr16,%fr16 /* %fr16 = r */	%fr16 = [r] ~512
L513	fcnvxf %fr17,%fr17 /* %fr17 = g */	%fr17 = [g] ~513
L514	fcnvxf %fr18,%fr18 /* %fr18 = b */	%fr18 = [b] ~514

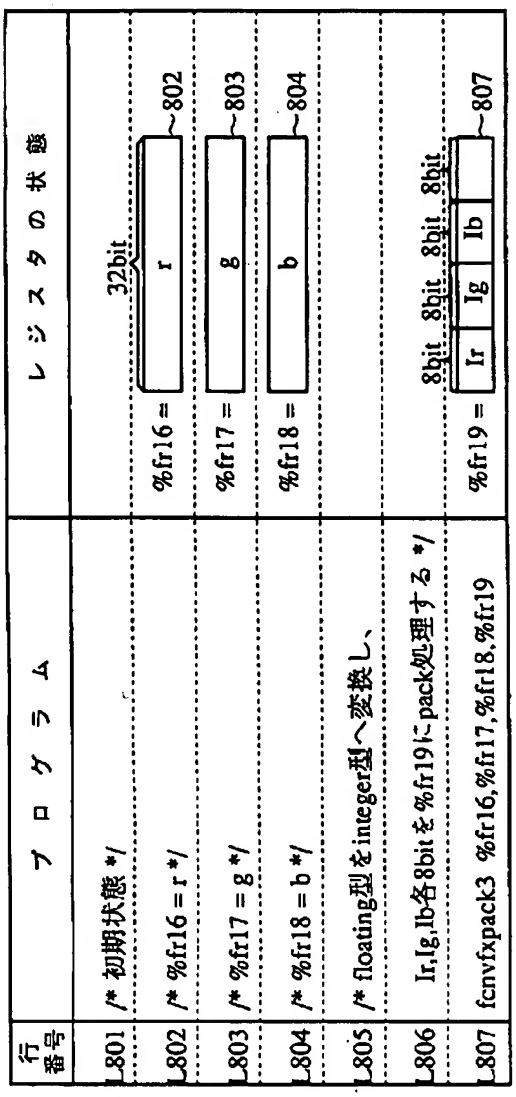
【図6】

図　6


行番号
プログラム
レジスタの状態
L601 /* 初期状態 */
32bit
L602 /* %fr16 = r */
%fr16 = r
~602
L603 /* %fr17 = g */
%fr17 = g
~603
L604 /* %fr18 = b */
%fr18 = b
~604
L605 /* floating型をinteger型へ変換し、
L606 Ir,Ig,Ibを各8bitを%fr19にpack処理する */
8bit 8bit 8bit 8bit
L607 fcnvfxdep %fr16, 7,8,%fr19
%fr19 = Ir
~607
L608 fcnvfxdep %fr17,15,8,%fr19
%fr19 = Ir Ig
~608
L609 fcnvfxdep %fr18,23,8,%fr19
%fr19 = Ir Ig Ib
~609

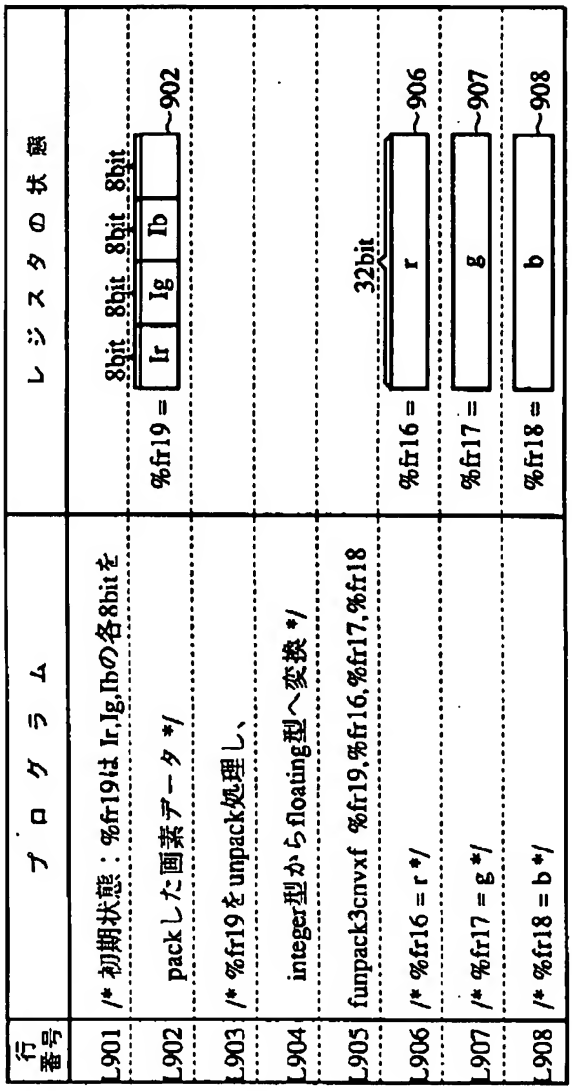
【図8】

図　8



【図9】

図　9



フロントページの続き

(72)発明者　藤井　秀樹
茨城県日立市大みか町五丁目２番１号　株
式会社日立製作所大みか工場内

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 7/00, 7/38, 7/52, 7/50, 9/30

A1

(11) International Publication Number: WO 97/08608

(43) International Publication Date: 6 March 1997 (06.03.97)

(21) International Application Number: PCT/US96/11893

(22) International Filing Date: 17 July 1996 (17.07.96)

(30) Priority Data:
08/521,360 31 August 1995 (31.08.95) US

(71) Applicant *(for all designated States except US)*: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: PELEG, Alexander, D. [IL/IL]; 38 Hannah Street, Carmelia, Haifa (IL). YAARI, Yaacov [IL/IL]; 17/2 Soerot Hanadin, Hanadin, Haifa (IL). MITTAL, Millind [IN/US]; 1149 Hillside Boulevard, S. San Francisco, CA (US). MENNEMEIER, Larry, M. [US/US]; P.O. Box 587, Boulder Creek, CA 95006 (US). EITAN, Benny [IL/IL]; 25 Stephen Wise, Haifa (IL). GLEW, Andrew, F. [CA/US]; 825 N.E. Kathryn, Hillsboro, OR 97124 (US). DULONG, Carole [FR/US]; 18983 Harleigh Drive, Saratoga, CA 95070 (US). KOWASHI, Eiichi [JP/US]; Apartment 618, 355 N. Wolfe Road, Sunnyvale, CA 94086 (US). WITT, Wolf [DE/US]; 2622 San Antonio Drive, Walnut Creek, CA 94598 (US).

(74) Agents: DE VOS, Daniel, M. et al.; Blakely, Sokoloff, Taylor & Zafman, 7th floor, 12400 Wilshire Boulevard, Los Angeles, CA 90025 (US).

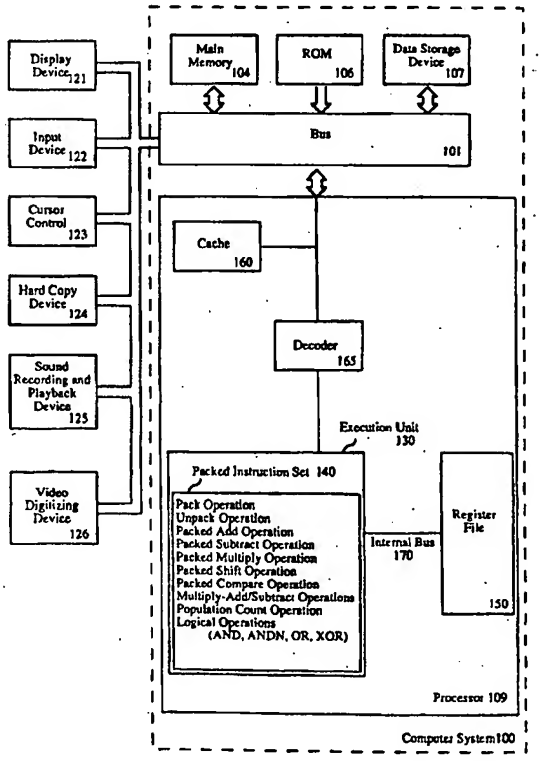
(81) Designated States: AL, AM, AT, AT (Utility model), AU, AZ, BB, BG, BR, BY, CA, CH, CN, CU, CZ, CZ (Utility model), DE, DE (Utility model), DK, DK (Utility model), EE, EE (Utility model), ES, FI, FI (Utility model), GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (Utility model), TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
With amended claims.

(54) Title: A SET OF INSTRUCTIONS FOR OPERATING ON PACKED DATA

(57) Abstract

An apparatus for including in a processor a set of instructions that support operations on packed data required by typical multimedia applications. In one embodiment, the invention includes a processor having a storage area (150), a decoder (165), and a plurality of circuits (130). The plurality of circuits provide for the execution of a number of instructions to manipulate packed data. In this embodiment, these instructions include pack, unpack, packed multiply, packed add, packed subtract, packed compare, and packed shift.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

A SET OF INSTRUCTIONS FOR OPERATING ON PACKED DATA

BACKGROUND OF THE INVENTION

1. FIELD OF INVENTION

In particular, the invention relates to the field of computer systems. More specifically, the invention relates to the area of packed data operations.

2. DESCRIPTION OF RELATED ART

In typical computer systems, processors are implemented to operate on values represented by a large number of bits (e.g., 64) using instructions that produce one result. For example, the execution of an add instruction will add together a first 64-bit value and a second 64-bit value and store the result as a third 64-bit value. However, multimedia applications (e.g., applications targeted at computer supported cooperation (CSC -- the integration of teleconferencing with mixed media data manipulation), 2D/3D graphics, image processing, video compression/decompression, recognition algorithms and audio manipulation) require the manipulation of large amounts of data which may be represented in a small number of bits. For example, graphical data typically requires 8 or 16 bits and sound data typically requires 8 or 16 bits. Each of these multimedia application requires one or more algorithms, each requiring a number of operations. For example, an algorithm may require an add, compare and shift operation.

To improve efficiency of multimedia applications (as well as other applications that have the same characteristics), prior art processors provide packed data formats. A packed data format is one in which the bits typically used to represent a single value are broken into a number of fixed sized data elements, each of which represents a separate value. For example, a 64-bit register may be broken into two 32-bit elements, each of which represents a separate 32-bit value. In addition, these prior art processors provide instructions for separately manipulating each element in these packed data types in parallel. For example, a packed add instruction adds together corresponding data elements

from a first packed data and a second packed data. Thus, if a multimedia algorithm requires a loop containing five operations that must be performed on a large number of data elements, it is desirable to pack the data and perform these operations in parallel using packed data instructions. In this manner, these processors can more efficiently process multimedia applications.

However, if the loop of operations contains an operation that cannot be performed by the processor on packed data (i.e., the processor lacks the appropriate instruction), the data will have to be unpacked to perform the operation. For example, if the multimedia algorithm required an add operation and the previously described packed add instruction is not available, the programmer must unpack both the first packed data and the second packed data (i.e., separate the elements comprising both the first packed data and the second packed data), add the separated elements together individually, and then pack the results into a packed result for further packed processing. The processing time required to perform such packing and unpacking often negates the performance advantage for which packed data formats are provided. Therefore, it is desirable to incorporate on a general purpose processor a set of packed data instructions that provide all the required operations for typical multimedia algorithms. However, due to the limited die area on today's microprocessors, the number of instructions which may be added is limited.

One general purpose processor that contains packed data instructions is the i860XP™ processor manufactured by Intel Corporation of Santa Clara, California. The i860XP processor includes several packed data types having different element sizes. In addition, the i860XP processor includes a packed add and a packed compare instruction. However, the packed add instruction does not break the carry chain, and thus the programmer has to insure the operations being performed by the software will not cause an overflow -- i.e., the operation will not cause bits from one element in the packed data to overflow into the next element of the packed data. For example, if a value of 1 is added to an 8-bit packed data element storing "11111111", an overflow occurs and the result is "1 00000000". In addition, the location of the decimal point in the packed data types supported by the i860XP processor is fixed (i.e., the i860XP processor supported 8.8, 6.10, and 8.24 numbers, where an i.j number contains the i most significant bits and j bits after the decimal point). Thus, the programmer is

limited as to the values that may be represented. Since the i860XP processor supports only these two instructions, it cannot perform many of the operations required by multimedia algorithms using packed data.

Another general purpose processor that supports packed data is the MC88110™ processor manufactured by Motorola, Inc. The MC88110 processor supports several different packed data formats having different size elements. In addition, the set of packed instructions supported by the MC88110 processor includes a pack, an unpack, a packed add, a packed subtract, a packed multiply, a packed compare, and a packed rotate.

The MC88110 processor pack command operates by concatenating the (t * r)/64 (where t is the number of bits in the elements of the packed data) most significant bits of each of the elements in a first register pair to generate a field of width r. This field replaces the most significant bits of the packed data stored in a second register pair. This packed data is then stored in a third register pair and rotated left by r bits. The table of supported values for t & r, as well as an example of the operation of this instruction, are shown in Tables 1 and 2 below.

		r		
		8	16	32
t	8	x	x	4
	16	x	4	8
	32	4	8	16

x = undefined operation

Table 1

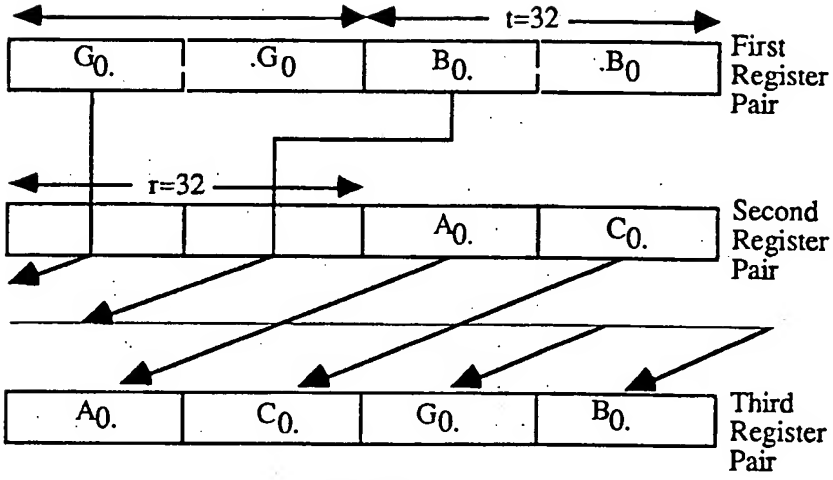


Table 2

This implementation of a pack instruction has two disadvantages. The first is that additional logic is required to perform the rotate at the end of the instruction.

The second is the number of instructions required to generate a packed data result. For example, if it is desired to use four 32-bit values to generate the result in the third register (shown above), 2 instructions with t=32 and r=32 are required as shown below in Table 3.

ppack Source1,Source2

A0.	.A0	C0.	.C0	Source1
x	x	x	x	Source2
		=		
x	x	A0.	C0.	Result1

ppack Result1,Source3

G0.	.G0	B0.	.B0	Result1
x	x	A0.	C0.	Source3
		=		
A0.	C0.	G0.	B0.	Result2

Table 3

The MC88110 processor unpack command operates by placing 4-, 8-, or 16-bit data elements from a packed data into the lower half of data elements that are twice as large (8, 16, or 32 bits) with zero fill -- i.e., the higher order bits in the resulting data elements are set to zero. An example of the operation of this unpack command is shown below in Table 4.

First Register Pair

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
7	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Unpack

Second Register Pair

00000000 10000000	00000000 01110000	00000000 10001111	00000000 10001000
3	*2*	*1*	*0*

Table 4

The MC88110 processor packed multiply instruction multiplies each element of a 64-bit packed data by a 32-bit value as if the packed data represented a single value as shown below in Table 5.

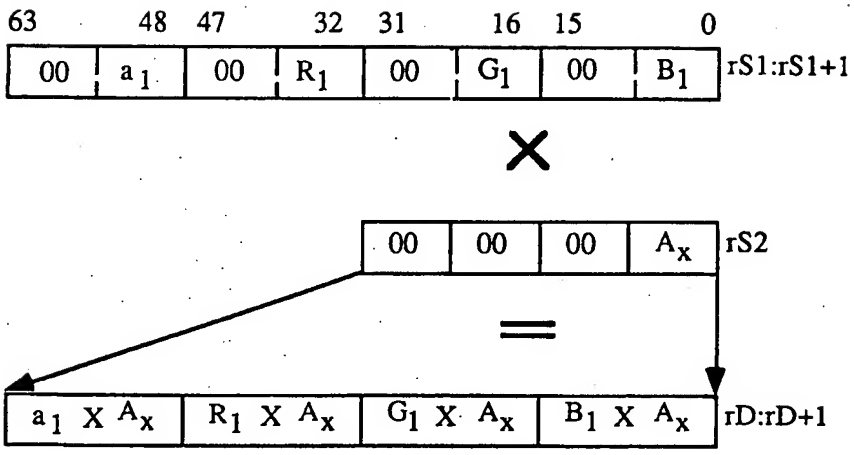
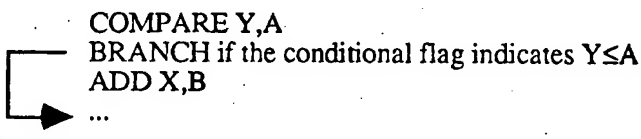


Table 5

This multiply instruction has two draw backs. First, this multiply instruction does not break the carry chains, and thus the programmer must insure that the operations performed on the packed data do not cause an overflow. As a result, the programmer is sometimes required to include additional instructions to prevent this overflow. Second, this multiply instruction multiplies each element in the packed data by a single value (i.e., the 32-bit value). As a result, the user does not have the flexibility to choose which elements in a packed data are multiplied by the 32-bit value. Therefore, the programmer must either prepare the data such that the same multiplications are required on every element in the packed data or waste processing time unpacking the data whenever a multiplication on less than all of the elements in the data is required. Thus, the programmer could not perform multiple multiplications using multiple

multipliers in parallel. For example, to multiply together eight different pieces of data, each one word long, requires four separate multiply operations. Each operation multiplying two words at a time, effectively wasting data lines and circuitry used for bits higher than bit sixteen.

The MC88110 processor packed compare instruction compares corresponding 32-bit data elements from a first packed data and a second packed data. Each of the two comparisons may return either less-than (<) or greater-than-or-equal-to (≥), resulting in four possible combinations. The instruction returns an 8-bit result string; four bits indicate which of the four possible conditions was met, and four bits indicate the complement of those bits. Conditional branching on the results of this instruction can be implemented in two ways: 1) with a sequence of conditional branches; or 2) with a jump table. A problem with this instruction is the fact that it requires conditional branches based on data to perform functions such as: if Y>A then X=X+B else X=X. A pseudo code compiled representation of this function would be:



New microprocessors try to speed up execution by speculatively predicting where branches go. If a prediction is correct, performance is not lost and there is a potential for a gain in performance. However, if a prediction is wrong, performance is lost. Therefore, the incentive to predict well is great. However, branches based on data (such as the one above) behave in an unpredictable way that breaks the prediction algorithms and results in more wrong predictions. As a result, use of this compare instruction to set up conditional branches on data comes at a high cost to performance.

The MC88110 processor rotate instruction rotates a 64-bit value to any modulo-4 boundary between 0 and 60 bits. (See Table 6 below for an example).

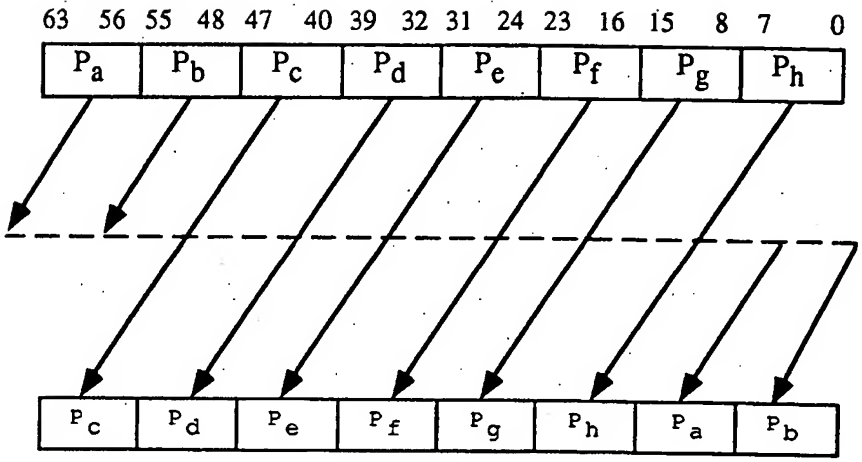


Table 6

Since the rotate instruction causes the high order bits that are shifted out of the register to be shifted into the low order bits of the register, the MC88110 processor does not support individually shifting each element in a packed data. As a result, programming algorithms which require individually shifting each element in a packed data type requires: 1) unpacking the data, 2) performing a shift on each element individually, and 3) packing the results into a result packed data for further packed data processing.

SUMMARY OF THE INVENTION

A method and apparatus for including in a processor a set of packed data instructions that support the operations required by typical multimedia applications is described. In one embodiment, the invention includes a processor and a storage area. The storage area contains a number of instructions for execution by the processor to manipulate packed data. In this embodiment, these instructions include pack, unpack, packed add, packed subtract, packed multiply, packed shift, and packed compare.

The processor packs a portion of the bits from data elements in at least two packed data to form a third packed data in response to receiving the pack instruction. In contrast, the processor generates a fourth packed data containing at least one data element from a first packed data operand and at least one corresponding data element from a second packed data operand in response to receiving the unpack instruction.

The processor separately adds together in parallel corresponding data elements from at least two packed data in response to receiving the packed add instruction. In contrast, the processor separately subtracts in parallel corresponding data elements from at least two packed data in response to receiving the packed subtract instruction.

The processor separately multiplies together in parallel corresponding data elements from at least two packed data in response to receiving the packed multiply instruction.

The processor separately shifts in parallel each of the data elements in a packed data operand by an indicated count in response to receiving the packed shift instruction.

The processor separately compares in parallel corresponding data elements from at least two packed data according to an indicated relationship and stores as a result a packed mask in a first register in response to receiving the packed compare instruction. The packed mask contains at least a first mask element and a second mask element. Each bit in the first mask element indicates the result of comparing one set of corresponding data elements, while each bit in the second mask element indicates the result of comparing a second set of data elements.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is illustrated by way of example, and not limitation, in the figures. Like references indicate similar elements.

Figure 1 illustrates an exemplary computer system according to one embodiment of the invention.

Figure 2 illustrates a register file of the processor according to one embodiment of the invention.

Figure 3 is a flow diagram illustrating the general steps used by the processor to manipulate data according to one embodiment of the invention.

Figure 4 illustrates packed data-types according to one embodiment of the invention.

Figure 5a illustrates in-register packed data representations according to one embodiment of the invention.

Figure 5b illustrates in-register packed data representations according to one embodiment of the invention.

Figure 5c illustrates in-register packed data representations according to one embodiment of the invention.

Figure 6a illustrates a control signal format for indicating the use of packed data according to one embodiment of the invention.

Figure 6b illustrates a second control signal format for indicating the use of packed data according to one embodiment of the invention.

PACKED ADD/SUBTRACT

Figure 7a illustrates a method for performing packed addition according to one embodiment of the invention.

Figure 7b illustrates a method for performing packed subtraction according to one embodiment of the invention.

Figure 8 illustrates a circuit for performing packed addition and packed subtraction on individual bits of packed data according to one embodiment of the invention.

Figure 9 illustrates a circuit for performing packed addition and packed subtraction on packed byte data according to one embodiment of the invention.

Figure 10 is a logical view of a circuit for performing packed addition and packed subtraction on packed word data according to one embodiment of the invention.

Figure 11 is a logical view of a circuit for performing packed addition and packed subtraction on packed doubleword data according to one embodiment of the invention.

PACKED MULTIPLY

Figure 12 is a flow diagram illustrating a method for performing packed multiplication operations on packed data according to one embodiment of the invention.

Figure 13 illustrates a circuit for performing packed multiplication according to one embodiment of the invention.

MULTIPLY-ADD/SUBTRACT

Figure 14 is a flow diagram illustrating a method for performing multiply-add and multiply-subtract operations on packed data according to one embodiment of the invention.

Figure 15 illustrates a circuit for performing multiply-add and/or multiply-subtract operations on packed data according to one embodiment of the invention.

PACKED SHIFT

Figure 16 is a flow diagram illustrating a method for performing a packed shift operation on packed data according to one embodiment of the invention.

Figure 17 illustrates a circuit for performing a packed shift on individual bytes of packed data according to one embodiment of the invention.

PACK

Figure 18 is a flow diagram illustrating a method for performing pack operations on packed data according to one embodiment of the invention.

Figure 19a illustrates a circuit for performing pack operations on packed byte data according to one embodiment of the invention.

Figure 19b illustrates a circuit for performing pack operations on packed word data according to one embodiment of the invention.

UNPACK

Figure 20 is a flow diagram illustrating a method for performing unpack operations on packed data according to one embodiment of the invention.

Figure 21 illustrates a circuit for performing unpack operations on packed data according to one embodiment of the invention.

POPULATION COUNT

Figure 22 is a flow diagram illustrating a method for performing a population count operation on packed data according to one embodiment of the invention.

Figure 23 is a flow diagram illustrating a method for performing a population count operation on one data element of a packed data and generating a single result data element for a result packed data according to one embodiment of the invention.

Figure 24 illustrates a circuit for performing a population count operation on packed data having four word data elements according to one embodiment of the invention.

Figure 25 illustrates a detailed circuit for performing a population count operation on one word data element of a packed data according to one embodiment of the invention.

PACKED LOGICAL OPERATIONS

Figure 26 is a flow diagram illustrating a method for performing a number of logical operations on packed data according to one embodiment of the invention.

Figure 27 illustrates a circuit for performing logical operations on packed data according to one embodiment of the invention.

PACKED COMPARE

Figure 28 is a flow diagram illustrating a method for performing packed compare operations on packed data according to one embodiment of the invention.

Figure 29 illustrates a circuit for performing packed compare operations on individual bytes of packed data according to one embodiment of the invention.

DETAILED DESCRIPTION

This application describes a method and apparatus for including in a processor a set of instructions that support the operations on packed data required by typical multimedia applications. In the following description, numerous specific details are set forth to provide a thorough understanding of the invention. However, it is understood that the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to unnecessarily obscure the invention.

DEFINITIONS

To provide a foundation for understanding the description of the embodiments of the invention, the following definitions are provided.

Bit X through Bit Y:

defines a subfield of binary number. For example, bit six through bit zero of the byte 001110102 (shown in base two) represent the subfield 1110102. The '2' following a binary number indicates base 2. Therefore, 10002 equals 810, while $F16$ equals 1510.

R_X: is a register. A register is any device capable of storing and providing data. Further functionality of a register is described below. A register is not necessarily part of the processor's package.

SRC1, SRC2, and DEST:

identify storage areas (e.g., memory addresses, registers, etc.)

Source1-i and Result1-i:

represent data.

COMPUTER SYSTEM

Figure 1 illustrates an exemplary computer system 100 according to one embodiment of the invention. Computer system 100 includes a bus 101, or other communications hardware and software, for communicating information, and a processor 109 coupled with bus 101 for processing information. Processor 109 represents a central processing unit of any type of architecture, including a CISC or RISC type architecture. Computer system 100 further includes a random access memory (RAM) or other dynamic storage device (referred to as main memory 104), coupled to bus 101 for storing information and instructions to be executed by processor 109. Main memory 104 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 109. Computer system 100 also includes a read only memory (ROM) 106, and/or other static storage device, coupled to bus 101 for storing static information and instructions for processor 109. Data storage device 107 is coupled to bus 101 for storing information and instructions.

Figure 1 also illustrates that processor 109 includes an execution unit 130, a register file 150, a cache 160, a decoder 165, and an internal bus 170. Of course, processor 109 contains additional circuitry which is not shown so as to not obscure the invention.

Execution unit 130 is used for executing instructions received by processor 109. In addition to recognizing instructions typically implemented in general purpose processors, execution unit 130 recognizes instructions in packed instruction set 140 for performing operations on packed data formats. In one embodiment, packed instruction set 140 includes instructions for supporting pack operation(s), unpack operation(s), packed add operation(s), packed subtract operation(s), packed multiply operation(s), packed shift operation(s), packed compare operation(s), multiply-add operation(s), multiply-subtract operation(s), population count operation(s), and a set of packed logical operations (including packed AND, packed ANDNOT, packed OR, and packed XOR) in the manner later described herein. While one embodiment is described in which packed instruction set 140 includes these instructions, alternative embodiment may contain a subset or a super-set of these instructions.

By including these instructions, the operations required by many of the algorithms used in multimedia applications may be performed using packed data. Thus, these algorithms may be written to pack the necessary data and perform the necessary operations on the packed data, without requiring the packed data to be unpacked to perform one or more operations one data element at a time. As previously described, this provides performance advantages over prior art general purpose processors that do not support the packed data operations required by certain multimedia algorithms -- i.e., if a multimedia algorithm requires an operation that cannot be performed on packed data, the program must unpack the data, perform the operation on the separate elements individually, and then pack the results into a packed result for further packed processing. In addition, the disclosed manner in which several of these instructions are performed improves the performance of many multimedia applications.

Execution unit 130 is coupled to register file 150 by internal bus 170. Register file 150 represents a storage area on processor 109 for storing information, including data. It is understood that one aspect of the invention is the described instruction set for operating on packed data. According to this

aspect of the invention, the storage area used for storing the packed data is not critical. However, one embodiment of the register file 150 is later described with reference to Figure 2. Execution unit 130 is coupled to cache 160 and decoder 165. Cache 160 is used to cache data and/or control signals from, for example, main memory 104. Decoder 165 is used for decoding instructions received by processor 109 into control signals and/or microcode entry points. In response to these control signals and/or microcode entry points, execution unit 130 performs the appropriate operations. For example, if an add instruction is received, decoder 165 causes execution unit 130 to perform the required addition; if a subtract instruction is received, decoder 165 causes execution unit 130 to perform the required subtraction; etc. Decoder 165 may be implemented using any number of different mechanisms (e.g., a look-up table, a hardware implementation, a PLA, etc.). Thus, while the execution of the various instructions by the decoder and execution unit is represented by a series of if/then statements, it is understood that the execution of an instruction does not require a serial processing of these if/then statements. Rather, any mechanism for logically performing this if/then processing is considered to be within the scope of the invention.

Figure 1 additionally shows a data storage device 107, such as a magnetic disk or optical disk, and its corresponding disk drive. Computer system 100 can also be coupled via bus 101 to a display device 121 for displaying information to a computer user. Display device 121 can include a frame buffer, specialized graphics rendering devices, a cathode ray tube (CRT), and/or a flat panel display. An alphanumeric input device 122, including alphanumeric and other keys, is typically coupled to bus 101 for communicating information and command selections to processor 109. Another type of user input device is cursor control 123, such as a mouse, a trackball, a pen, a touch screen, or cursor direction keys for communicating direction information and command selections to processor 109, and for controlling cursor movement on display device 121. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), which allows the device to specify positions in a plane. However, this invention should not be limited to input devices with only two degrees of freedom.

Another device which may be coupled to bus 101 is a hard copy device 124 which may be used for printing instructions, data, or other information on a medium such as paper, film, or similar types of media. Additionally, computer system 100 can be coupled to a device for sound recording, and/or playback 125, such as an audio digitizer coupled to a microphone for recording information. Further, the device may include a speaker which is coupled to a digital to analog (D/A) converter for playing back the digitized sounds.

Also, computer system 100 can be a terminal in a computer network (e.g., a LAN). Computer system 100 would then be a computer subsystem of a computer network. Computer system 100 optionally includes video digitizing device 126. Video digitizing device 126 can be used to capture video images that can be transmitted to others on the computer network.

In one embodiment, the processor 109 additionally supports an instruction set which is compatible with the x86 instruction set (the instruction set used by existing microprocessors, such as the Pentium® processor, manufactured by Intel Corporation of Santa Clara, California). Thus, in one embodiment, processor 109 supports all the operations supported in the IA™ - Intel Architecture, as defined by Intel Corporation of Santa Clara, California (see Microprocessors, Intel Data Books volume 1 and volume 2, 1992 and 1993, available from Intel of Santa Clara, California). As a result, processor 109 can support existing x86 operations in addition to the operations of the invention. While the invention is described as being incorporated into an x86 based instruction set, alternative embodiments could incorporate the invention into other instruction sets. For example, the invention could be incorporated into a 64-bit processor using a new instruction set.

Figure 2 illustrates the register file of the processor according to one embodiment of the invention. The register file 150 is used for storing information, including control/status information, integer data, floating point data, and packed data. In the embodiment shown in Figure 2, the register file 150 includes integer registers 201, registers 209, status registers 208, and instruction pointer register 211. Status registers 208 indicate the status of processor 109. Instruction pointer register 211 stores the address of the next instruction to be executed. Integer registers 201, registers 209, status registers

208, and instruction pointer register 211 are all coupled to internal bus 170. Any additional registers would also be coupled to internal bus 170.

In one embodiment, the registers 209 are used for both packed data and floating point data. In one such embodiment, the processor 109, at any given time, must treat the registers 209 as being either stack referenced floating point registers or non-stack referenced packed data registers. In this embodiment, a mechanism is included to allow the processor 109 to switch between operating on registers 209 as stack referenced floating point registers and non-stack referenced packed data registers. In another such embodiment, the processor 109 may simultaneously operate on registers 209 as non-stack referenced floating point and packed data registers. As another example, in another embodiment, these same registers may be used for storing integer data.

Of course, alternative embodiments may be implemented to contain more or less sets of registers. For example, an alternative embodiment may include a separate set of floating point registers for storing floating point data. As another example, an alternative embodiment may including a first set of registers, each for storing control/status information, and a second set of registers, each capable of storing integer, floating point, and packed data. As a matter of clarity, the registers of an embodiment should not be limited in meaning to a particular type of circuit. Rather, a register of an embodiment need only be capable of storing and providing data, and performing the functions described herein.

The various sets of registers (e.g., the integer registers 201, the registers 209) may be implemented to include different numbers of registers and/or to different size registers. For example, in one embodiment, the integer registers 201 are implemented to store thirty-two bits, while the registers 209 are implemented to store eighty bits (all eighty bits are used for storing floating point data, while only sixty-four are used for packed data). In addition, registers 209 contains eight registers, R_0 212a through R_7 212h. R_1 212a, R_2 212b and R_3 212c are examples of individual registers in registers 209. Thirty-two bits of a register in registers 209 can be moved into an integer register in integer registers 201. Similarly, a value in an integer register can be moved into thirty-two bits of a register in registers 209. In another embodiment, the integer registers 201 each contain 64 bits, and 64 bits of data may be moved between the integer register 201 and the registers 209.

Figure 3 is a flow diagram illustrating the general steps used by the processor to manipulate data according to one embodiment of the invention. For example, such operations include a load operation to load a register in register file 150 with data from cache 160, main memory 104, read only memory (ROM) 106, or data storage device 107.

At step 301, the decoder 202 receives a control signal 207 from either the cache 160 or bus 101. Decoder 202 decodes the control signal to determine the operations to be performed.

At step 302, Decoder 202 accesses the register file 150, or a location in memory. Registers in the register file 150, or memory locations in the memory, are accessed depending on the register address specified in the control signal 207. For example, for an operation on packed data, control signal 207 can include SRC1, SRC2 and DEST register addresses. SRC1 is the address of the first source register. SRC2 is the address of the second source register. In some cases, the SRC2 address is optional as not all operations require two source addresses. If the SRC2 address is not required for an operation, then only the SRC1 address is used. DEST is the address of the destination register where the result data is stored. In one embodiment, SRC1 or SRC2 is also used as DEST. SRC1, SRC2 and DEST are described more fully in relation to Figure 6a and Figure 6b. The data stored in the corresponding registers is referred to as Source1, Source2, and Result respectively. Each of these data is sixty-four bits in length.

In another embodiment of the invention, any one, or all, of SRC1, SRC2 and DEST, can define a memory location in the addressable memory space of processor 109. For example, SRC1 may identify a memory location in main memory 104, while SRC2 identifies a first register in integer registers 201 and DEST identifies a second register in registers 209. For simplicity of the description herein, the invention will be described in relation to accessing the register file 150. However, these accesses could be made to memory instead.

At step 303, execution unit 130 is enabled to perform the operation on the accessed data. At step 304, the result is stored back into register file 150 according to requirements of control signal 207.

DATA AND STORAGE FORMATS

Figure 4 illustrates packed data-types according to one embodiment of the invention. Three packed data formats are illustrated; packed byte 401, packed word 402, and packed doubleword 403. Packed byte, in one embodiment of the invention, is sixty-four bits long containing eight data elements. Each data element is one byte long. Generally, a data element is an individual piece of data that is stored in a single register (or memory location) with other data elements of the same length. In one embodiment of the invention, the number of data elements stored in a register is sixty-four bits divided by the length in bits of a data element.

Packed word 402 is sixty-four bits long and contains four word 402 data elements. Each word 402 data element contains sixteen bits of information.

Packed doubleword 403 is sixty-four bits long and contains two doubleword 403 data elements. Each doubleword 403 data element contains thirty-two bits of information.

Figure 5a through 5c illustrate the in-register packed data storage representation according to one embodiment of the invention. Unsigned packed byte in-register representation 510 illustrates the storage of an unsigned packed byte 401 in one of the registers R_0 212a through R_7 212h. Information for each byte data element is stored in bit seven through bit zero for byte zero, bit fifteen through bit eight for byte one, bit twenty-three through bit sixteen for byte two, bit thirty-one through bit twenty-four for byte three, bit thirty-nine through bit thirty-two for byte four, bit forty-seven through bit forty for byte five, bit fifty-five through bit forty-eight for byte six and bit sixty-three through bit fifty-six for byte seven. Thus, all available bits are used in the register. This storage arrangement increases the storage efficiency of the processor. As well, with eight data elements accessed, one operation can now be performed on eight data elements simultaneously. Signed packed byte in-register representation 511 illustrates the storage of a signed packed byte 401. Note that only the eighth bit of every byte data element is necessary for the sign indicator.

Unsigned packed word in-register representation 512 illustrates how word three through word zero are stored in one register of registers 209. Bit fifteen through bit zero contain the data element information for word zero, bit thirty-one through bit sixteen contain the information for data element word one, bit

forty-seven through bit thirty-two contain the information for data element word two and bit sixty-three through bit forty-eight contain the information for data element word three. Signed packed word in-register representation 513 is similar to the unsigned packed word in-register representation 512. Note that only the sixteenth bit of each word data element is the necessary for the sign indicator.

Unsigned packed doubleword in-register representation 514 shows how registers 209 store two doubleword data elements. Doubleword zero is stored in bit thirty-one through bit zero of the register. Doubleword one is stored in bit sixty-three through bit thirty-two of the register. Signed packed doubleword in-register representation 515 is similar to unsigned packed doubleword in-register representation 514. Note that the necessary sign bit is the thirty-second bit of the doubleword data element.

As mentioned previously, registers 209 may be used for both packed data and floating point data. In this embodiment of the invention, the individual programming processor 109 may be required to track whether an addressed register, R_0 212a for example, is storing packed data or floating point data. In an alternative embodiment, processor 109 could track the type of data stored in individual registers of registers 209. This alternative embodiment could then generate errors if, for example, a packed addition operation were attempted on floating point data.

CONTROL SIGNAL FORMATS

The following describes one embodiment of control signal formats used by processor 109 to manipulate packed data. In one embodiment of the invention, control signals are represented as thirty-two bits. Decoder 202 may receive control signal 207 from bus 101. In another embodiment, decoder 202 can also receive such control signals from cache 160.

Figure 6a illustrates a control signal format for indicating the use of packed data according to one embodiment of the invention. Operation field OP 601, bit thirty-one through bit twenty-six, provides information about the operation to be performed by processor 109; for example, packed addition, packed subtraction, etc.. SRC1 602, bit twenty-five through twenty, provides the source register address of a register in registers 209. This source register contains the first packed data, Source1, to be used in the execution of the control signal. Similarly,

SRC2 603, bit nineteen through bit fourteen, contains the address of a register in registers 209. This second source register contains the packed data, Source2, to be used during execution of the operation. DEST 605, bit five through bit zero, contains the address of a register in registers 209. This destination register will store the result packed data, Result, of the packed data operation.

Control bits SZ 610, bit twelve and bit thirteen, indicates the length of the data elements in the first and second packed data source registers. If SZ 610 equals 01_2, then the packed data is formatted as packed byte 401. If SZ 610 equals 10_2, then the packed data is formatted as packed word 402. SZ 610 equaling 00_2 or 11_2 is reserved, however, in another embodiment, one of these values could be used to indicate packed doubleword 403.

Control bit T 611, bit eleven, indicates whether the operation is to be carried out with saturate mode. If T 611 equals one, then a saturating operation is performed. If T 611 equals zero, then a non-saturating operation is performed. Saturating operations will be described later.

Control bit S 612, bit ten, indicates the use of a signed operation. If S 612 equals one, then a signed operation is performed. If S 612 equals zero, then an unsigned operation is performed.

Figure 6b illustrates a second control signal format for indicating the use of packed data according to one embodiment of the invention. This format corresponds with the general integer opcode format described in the "Pentium Processor Family User's Manual," available from Intel Corporation, Literature Sales, P.O. Box 7641, Mt. prospect, IL, 60056-7641. Note that OP 601, SZ 610, T 611, and S 612 are all combined into one large field. For some control signals, bits three through five are SRC1 602. In one embodiment, where there is a SRC1 602 address, then bits three through five also correspond to DEST 605. In an alternate embodiment, where there is a SRC2 603 address, then bits zero through two also correspond to DEST 605. For other control signals, like a packed shift immediate operation, bits three through five represent an extension to the opcode field. In one embodiment, this extension allows a programmer to include an immediate value with the control signal, such as a shift count value. In one embodiment, the immediate value follows the control signal. This is described in more detail in the "Pentium Processor Family User's Manual," in appendix F, pages F-1 through F-3. Bits zero through two represent SRC2 603. This general

format allows register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing. Also, in one embodiment, this general format can support integer register to register, and register to integer register addressing.

DESCRIPTION OF SATURATE/UNSATURATE

As mentioned previously, T 611 indicates whether operations optionally saturate. Where the result of an operation, with saturate enabled, overflows or underflows the range of the data, the result will be clamped. Clamping means setting the result to a maximum or minimum value should a result exceed the range's maximum or minimum value. In the case of underflow, saturation clamps the result to the lowest value in the range and in the case of overflow, to the highest value. The allowable range for each data format is shown in Table 7.

Data Format	Minimum Value	Maximum Value
Unsigned Byte	0	255
Signed Byte	-128	127
Unsigned Word	0	65535
Signed Word	-32768	32767
Unsigned Doubleword	0	$2^{64}-1$
Signed Doubleword	-2^{63}	$2^{63}-1$

Table 7

As mentioned above, T 611 indicates whether saturating operations are being performed. Therefore, using the unsigned byte data format, if an operation's result = 258 and saturation was enabled, then the result would be clamped to 255 before being stored into the operation's destination register. Similarly, if an operation's result = -32999 and processor 109 used signed word data format with saturation enabled, then the result would be clamped to -32768 before being stored into the operation's destination register.

PACKED ADDITION

PACKED ADDITION OPERATION

One embodiment of the invention enables packed addition operations to be performed in Execution unit 130. That is, the invention enables each data element of a first packed data to be added individually to each data element of a second packed data.

Figure 7a illustrates a method for performing packed addition according to one embodiment of the invention. At step 701, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for packed addition; SRC1 602, SRC2 603 and DEST 605 addresses in registers 209; saturate/unsaturate, signed/unsigned, and length of the data elements in the packed data. At step 702, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provides Execution unit 130 with the packed data stored in the registers at these addresses, Source1 and Source2 respectively. That is, registers 209 communicate the packed data to Execution unit 130 via internal bus 170.

At step 703, decoder 202 enables Execution unit 130 to perform a packed addition operation. Decoder 202 further communicates, via internal bus 170, the length of packed data elements, whether saturation is to be used, and whether signed arithmetic is to be used. At step 704, the length of the data element determines which step is to be executed next. If the length of the data elements in the packed data is eight bits (byte data), then Execution unit 130 performs step 705a. However, if the length of the data elements in the packed data is sixteen bits (word data), then Execution unit 130 performs step 705b. In one embodiment of the invention, only eight bit and sixteen bit data element length packed addition is supported. However, alternative embodiments can support different and/or other lengths. For example, an alternative embodiment could additionally support thirty-two bit data element length packed addition.

Assuming the length of the data elements is eight bits, then step 705a is executed. Execution unit 130 adds bit seven through bit zero of Source1 to bit seven through bit zero of SRC2, producing bit seven through bit zero of Result packed data. In parallel with this addition, Execution unit 130 adds bit fifteen through bit eight of Source1 to bit fifteen through bit eight of Source2, producing bit fifteen through bit eight of Result packed data. In parallel with these

additions, Execution unit 130 adds bit twenty-three through bit sixteen of Source1 to bit twenty-three through bit sixteen of Source2, producing bit twenty-three through bit sixteen of Result packed data. In parallel with these additions, Execution unit 130 adds bit thirty-one through bit twenty-four of Source1 to bit thirty-one through bit twenty-four of Source2, producing bit thirty-one through bit twenty-four of Result packed data. In parallel with these additions, Execution unit 130 adds bit thirty-nine through bit thirty-two of Source1 to bit thirty-nine through bit thirty-two of Source2, producing bit thirty-nine through bit thirty-two of Result packed data. In parallel with these additions, Execution unit 130 adds bit forty-seven through bit forty of Source1 to bit forty-seven through bit forty of Source2, producing bit forty-seven through bit forty of Result packed data. In parallel with these additions, Execution unit 130 adds bit fifty-five through bit forty-eight of Source1 to bit fifty-five through bit forty-eight of Source2, producing bit fifty-five through bit forty-eight of Result packed data. In parallel with these additions, Execution unit 130 adds bit sixty-three through bit fifty-six of Source1 to bit sixty-three through bit fifty-six of Source2, producing bit sixty-three through bit fifty-six of Result packed data.

Assuming the length of the data elements is sixteen bits, then step 705b is executed. Execution unit 130 adds bit fifteen through bit zero of Source1 to bit fifteen through bit zero of SRC2, producing bit fifteen through bit zero of Result packed data. In parallel with this addition, Execution unit 130 adds bit thirty-one through bit sixteen of Source1 to bit thirty-one through bit sixteen of Source2, producing bit thirty-one through bit sixteen of Result packed data. In parallel with these additions, Execution unit 130 adds bit forty-seven through bit thirty-two of Source1 to bit forty-seven through bit thirty-two of Source2, producing bit forty-seven through bit thirty-two of Result packed data. In parallel with these additions, Execution unit 130 adds bit sixty-three through bit forty-eight of Source1 to bit sixty-three through bit forty-eight of Source2, producing bit sixty-three through bit forty-eight of Result packed data.

At step 706, decoder 202 enables a register in registers 209 with DEST 605 address of the destination register. Thus, the Result is stored in the register addressed by DEST 605.

Table 8a illustrates the in-register representation of packed addition operation. The first row of bits is the packed data representation of a Source1

packed data. The second row of bits is the packed data representation of a Source2 packed data. The third row of bits is the packed data representation of the Result packed data. The number below each data element bit is the data element number. For example, Source1 data element 0 is 10001000_2. Therefore, if the data elements are eight bits in length (byte data), and unsigned, unsaturated addition is performed, the Execution unit 130 produces the Result packed data as shown.

Note that in one embodiment of the invention, where a result overflows or underflows and the operation is using unsaturate, that result is simply truncated. That is, the carry bit is ignored. For example, in Table 8a, the in-register representation of result data element one would be: $10001000_2 + 10001000_2 = 00001000_2$. Similarly, for underflows, the result is truncated. This form of truncation enables a programmer to easily perform module arithmetic. For example, an equation for result data element one can be expressed as: (Source1 data element one + Source2 data element one) mod 256 = result data element one. Further, one skilled in the art would understand from this description that overflows and underflows could be detected by setting error bits in a status register.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
+ 7	+ 6	+ 5	+ 4	+ 3	+ 2	+ 1	+ 0
10101010	01010101	10101010	10000001	10000000	11110000	11001111	10001000
= 7	= 6	= 5	= 4	= 3	= 2	= 1	= 0
11010100	10101010	11111111	Overflow	Overflow	Overflow	Overflow	Overflow
7	6	5	4	3	2	1	0

Table 8a

Table 8b illustrates the in-register representation of a packed word data addition operation. Therefore, if the data elements are sixteen bits in length (word data), and unsigned, unsaturated addition is performed, the Execution unit 130 produces the Result packed data as shown. Note that in word data element

two, the carry from bit seven (see emphasized bits *1* below) propagated into bit eight, causing data element two to overflow (see emphasized ***overflow*** below).

00101010 01010101	01010101 *1*1111111	10000000 01110000	10001111 10001000
+ 3	+ 2	+ 1	+ 0
10101010 01010101	10101010 *1*0000001	10000000 11110000	11001111 10001000
= 3	= 2	= 1	= 0
11010100 10101010	***Overflow***	Overflow	Overflow
3	2	1	0

Table 8b

Table 8c illustrates the in-register representation of packed doubleword data addition operation. This operation is supported in an alternative embodiment of the invention. Therefore, if the data elements are thirty-two bits in length (i.e., doubleword data), and unsigned, unsaturated addition is performed, the Execution unit 130 produces the Result packed data as shown. Note that carries from bit seven and bit fifteen of doubleword data element one propagated into bit eight and bit sixteen respectively.

00101010 01010101 01010101 11111111	10000000 01110000 10001111 10001000
+ 1	+ 0
10101010 01010101 10101010 10000001	10000000 11110000 11001111 10001000
= 1	= 0
11010100 10101011 00000000 10000000	Overflow
1	0

Table 8c

To better illustrate the difference between packed addition and ordinary addition, the data from the above example is duplicated in Table 9. However, in this case, ordinary addition (sixty-four bit) is performed on the data. Note that the carries from bit seven, bit fifteen, bit twenty-three, bit thirty-one, bit thirty-nine and bit forty-seven have been carried into bit eight, bit sixteen, bit twenty-four, bit thirty-two, bit forty and bit forty-eight respectively.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
±							
10101010	01010101	10101010	10000001	10000000	11110000	11001111	10001000
=							
11010100	10101011	00000000	10000001	00000001	01100001	01011111	00010000

Table 9

SIGNED/UNSATURATE PACKED ADDITION

Table 10 illustrates an example of a signed packed addition where the data element length of the packed data is eight bits. Saturation is not used. Therefore, results can overflow and underflow. Table 10 uses different data than Tables 8a–8c and Table 9.

00101010	01010101	01010101	01111111	00000000	11110000	00001111	10001000
+ 7	+ 6	+ 5	+ 4	+ 3	+ 2	+ 1	+ 0
10101010	01010101	10101010	00000001	00000000	11110000	00001111	10001000
= 7	= 6	= 5	= 4	= 3	= 2	= 1	= 0
11010100	Overflow	11111111	Overflow	00000000	Underflow	00011110	Underflow
7	6	5	4	3	2	1	0

Table 10

SIGNED/SATURATE PACKED ADDITION

Table 11 illustrates an example of a signed packed addition where the data element length of the packed data is eight bits. Saturate is used, therefore, overflow will be clamped to the maximum value, and underflow will be clamped to the minimum value. Table 11 uses the same data as Table 10. Here data

element zero and data element two are clamped to the minimum value, while data element four and data element six are clamped to the maximum value.

00101010	01010101	01010101	01111111	00000000	11110000	00001111	10001000
+ 7	+ 6	+ 5	+ 4	+ 3	+ 2	+ 1	+ 0
10101010	01010101	10101010	00000001	00000000	11110000	00001111	10001000
= 7	= 6	= 5	= 4	= 3	= 2	= 1	= 0
11010100	01111111	11111111	01111111	00000000	10000000	00011110	10000000
7	6	5	4	3	2	1	0

Table 11

PACKED SUBTRACTION

PACKED SUBTRACTION OPERATION

One embodiment of the invention enables packed subtraction operations to be performed in Execution unit 130. That is, the invention enables each data element of a second packed data to be subtracted individually from each data element of a first packed data.

Figure 7b illustrates a method for performing packed subtraction according to one embodiment of the invention. Note that steps 710-713 are similar to steps 701-704.

In the present embodiment of the invention, only eight bit and sixteen bit data element length packed subtraction is supported. However, alternative embodiments can support different and/or other lengths. For example, an alternative embodiment could additionally support, thirty-two bit data element length packed subtraction.

Assuming data element length is eight bits, steps 714a and 715a are executed. Execution unit 130 2's complements bit seven through bit zero of Source2. In parallel with this 2's complement, Execution unit 130 2's complements bit fifteen through bit eight of Source2. In parallel with these 2's

complements, Execution unit 130 2's complements bit twenty-three through bit sixteen of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit thirty-one through bit twenty-four of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit thirty-nine through bit thirty-two of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit forty-seven through bit forty of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit fifty-five through bit forty-eight of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit sixty-three through bit fifty-six of Source2. At step 715a, Execution unit 130 performs the addition of the 2's complemented bits of Source2 to the bits of Source1 as generally described for step 705a.

Assuming data element length is sixteen bits, steps 714b and 715b are executed. Execution unit 130 2's complements bit fifteen through bit zero of Source2. In parallel with this 2's complement, Execution unit 130 2's complements bit thirty-one through bit sixteen of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit forty-seven through bit thirty-two of Source2. In parallel with these 2's complements, Execution unit 130 2's complements bit sixty-three through bit forty-eight of Source2. At step 715b, Execution unit 130 performs the addition of the 2's complemented bits of Source2 to the bits of Source1 as generally described for step 705b.

Note that steps 714 and 715 are the method used in one embodiment of the invention to subtract a first number from a second number. However, other forms of subtraction are known in the art and this invention should not be considered limited to using 2's complement arithmetic.

At step 716, decoder 202 enables registers 209 with the destination address of the destination register. Thus, the result packed data is stored in the DEST register of registers 209.

Table 12 illustrates the in-register representation of packed subtraction operation. Assuming the data elements are eight bits in length (byte data), and unsigned, unsaturated subtraction is performed, then Execution unit 130 produces the result packed data as shown.

00101010	01010101	01010101	01111111	00000000	11110000	00001111	10001000
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-
10101010	01010101	10101010	00000001	00000000	11110000	00001111	10001000
7	6	5	4	3	2	1	0
=	=	=	=	=	=	=	=
Underflow	00000000	Underflow	01111110	00000000	00000000	00000000	00000000
7	6	5	4	3	2	1	0

Table 12

PACKED DATA ADDITION/SUBTRACTION CIRCUITS

Figure 8 illustrates a circuit for performing packed addition and packed subtraction on individual bits of packed data according to one embodiment of the invention. Figure 8 shows a modified bit slice adder/subtractor 800. Adder/subtractor 801a-b enable two bits from Source2 to be added to, or subtracted from, Source1. Operation and carry control 803 transmits to control 809a control signals to enable an addition or subtraction operation. Thus, adder/subtractor 801a adds or subtracts bit *i* received on Source2$_i$ 805a to bit *i* received on Source1$_i$ 804a, producing a result bit transmitted on Result$_i$ 806a. C_{in} 807a-b and C_{out} 808a-b represent carry control circuitry as is commonly found on adder/subtractors.

Bit control 802 is enabled from operation and carry control 803 via packed data enable 811 to control $C_{in_{i+1}}$ 807b and C_{out_i}. For example, in Table 13a, an unsigned packed byte addition is performed. If adder/subtractor 801a adds Source1 bit seven to Source2 bit seven, then operation and carry control 803 will enable bit control 802, stopping the propagation of a carry from bit seven to bit eight.

...	...	...	...	...	...	00001111	10001000
+ 7	+ 6	+ 5	+ 4	+ 3	+ 2	+ 1	+ 0
...	...	...	...	...	...	00001111	10001000
= 7	= 6	= 5	= 4	= 3	= 2	= 1	= 0
...	...	...	...	...	...	00011110	Overflow
7	6	5	4	3	2	1	0

Table 13a

However, if an unsigned packed word addition is performed, and adder/subtractor 801a is similarly used to add bit seven of Source1 to bit seven of Source2, bit control 802 propagates the carry to bit eight. Table 13b illustrates this result. This propagation would be allowed for packed doubleword addition as well as unpacked addition.

...	...	...	00001111 10001000
+ 3	+ 2	+ 1	+ 0
...	...	...	00001111 10001000
= 3	= 2	= 1	= 0
...	...	...	00011111 00010000
3	2	1	0

Table 13b

Adder/subtractor 801a subtracts bit $Source2_i$ 805a from $Source1_i$ 804a by first forming the 2's complement of $Source2_i$ 805a by inverting $Source2_i$ 805a

and adding one. Then adder/subtractor 801a adds this result to $Source1_i$ 804a. Bit slice 2's complementing techniques are well known in the art, and one skilled in the art would understand how to design such a bit slice 2's complementing circuit. Note that propagation of carries are controlled by bit control 802 and operation and carry control 803.

Figure 9 illustrates a circuit for performing packed addition and packed subtraction on packed byte data according to one embodiment of the invention. Source1 bus 901 and Source2 bus 902 carry the information signals to the adder/subtractors 908a-h via $Source1_{in}$ 906a-h and $Source2_{in}$ 905a-h respectively. Thus, adder/subtractor 908a adds/subtracts Source2 bit seven through bit zero to/from Source1 bit seven through bit zero; adder/subtractor 908b adds/subtracts Source2 bit fifteen through bit eight to/from Source1 bit fifteen through bit eight, etc.. CTRL 904a-h receives, from Operation Control 903, via packed control 911, control signals disabling the propagation of carries, enabling/disabling saturate, and enabling/disabling signed/unsigned arithmetic. Operation Control 903 disables propagation of carries by receiving carry information from CTRL 904a-h and not propagating it to the next most significant adder/subtractor 908a-h. Thus, Operation Control 903 performs the operations of the operation and carry control 803 and the bit control 802 for 64 bit packed data. One skilled in the art would be able create such a circuit given the illustrations in Figures 1-9 and the above description.

Adder/subtractors 908a-h communicate result information, via result out 907a-h, of the various packed additions to result register 910a-h. Each result register 910a-h stores and then transmits the result information onto Result bus 909. This result information is then stored in the integer register specified by the DEST 605 register address.

Figure 10 is a logical view of a circuit for performing packed addition and packed subtraction on packed word data according to one embodiment of the invention. Here, packed word operations are being performed. Propagation of carries between bit eight and bit seven, bit twenty-four and bit twenty-three, bit forty and bit thirty-nine, and bit fifty-six and bit fifty-five are enabled by Operation Control 903. Thus, adder/subtractor 908a and 908b, shown as virtual adder/subtractor 1008a, will act together to add/subtract the first word of packed word data Source2 (bit fifteen through bit zero) to/from the first word of packed

word data Source1 (bit fifteen through bit zero); adder/subtractor 908c and 908d, shown as virtual adder/subtractor 1008b, will act together to add/subtract the second word of packed word data Source2 (bit thirty-one through bit sixteen) to/from the second word of packed word data Source1 (bit thirty-one through bit sixteen), etc..

Virtual adder/subtractors 1008a-d communicate result information, via result out 1007a-d (combined result outs 907a-b, 907c-d, 907e-f and 907g-h), to virtual result registers 1010a-d. Each virtual result register 1010a-d (combined result registers 910a-b, 910c-d, 910e-f and 910g-h) stores a sixteen bit result data element to be communicated onto Result bus 909.

Figure 11 is a logical view of a circuit for performing packed addition and packed subtraction on packed doubleword data according to one embodiment of the invention. Propagation of carries between bit eight and bit seven, bit sixteen and bit fifteen, bit twenty-four and bit twenty-three, bit forty and bit thirty-nine, bit forty-eight and bit forty-seven, and bit fifty-six and bit fifty-five are enabled by Operation Control 903. Thus, adder/subtractors 908a-d, shown as virtual adder/subtractor 1108a, act together to add/subtract the first doubleword of packed doubleword data Source2 (bit thirty-one through bit zero) to/from the first doubleword of packed word data Source1 (bit thirty-one through bit zero); adder/subtractors 908e-h, shown as virtual adder/subtractor 1108b, act together to add/subtract the second doubleword of packed doubleword data Source2 (bit sixty-three through bit thirty-two) to/from the second doubleword of packed doubleword data Source1 (bit sixty-three through bit thirty-two).

Virtual adder/subtractors 1108a-b communicate result information, via result out 1107a-b (combined result outs 907a-d and 907e-h), to virtual result registers 1110a-b. Each virtual result register 1110a-b (combined result registers 910a-d and 910e-h) stores a thirty-two bit result data element to be communicated onto Result bus 909.

PACKED MULTIPLY

PACKED MULTIPLY OPERATION

In one embodiment of the invention, the SRC1 register contains multiplicand data (Source1), the SRC2 register contains multiplier data

(Source2), and DEST register will contain a portion of the product of the multiplication (Result). That is, Source1 will have each data element independently multiplied by the respective data element of Source2. Depending on the type of the multiply, the Result will include the high order or the low order bits of the product.

In one embodiment of the invention, the following multiply operations are supported: multiply high unsigned packed, multiply high signed packed and multiply low packed. Highllow indicate which bits from the product of the multiplication are to be included in the Result. This is needed because a multiplication of two N bit numbers results in a product having 2N bits. As each result data element is the same size as the multiplicand and the multiplier's data elements, only half of the product can be represented by the result. High causes the higher order bits to be output as the result. Low causes the low order bits to be output as the result. For example, unsigned high packed multiplication of Source1[7:0] by Source2[7:0] stores the high order bits of the product in Result[7:0].

In one embodiment of the invention, the use of the highllow operation modifier removes the possibility of an overflow from one data element into the next higher data element. That is, this modifier allows the programmer to select which bits of the product are to be in the result without concern for overflows. The programmer can generate a complete 2N bit product using a combination of packed multiply operations. For example, the programmer can use a multiply high unsigned packed operation and then, using the same Source1 and Source2, a multiply low packed operation to obtain complete (2N) products. The multiply high operation is provided because, often, the high order bits of the product are the only important part of the product. The programmer can obtain the high order bits of the product without first having to perform any truncation, as is often required by a nonpacked data operation.

In one embodiment of the invention, each data element in Source2 can have a different value. This provides the programmer with the flexibility to have a different value as the multiplier for each multiplicand in Source1.

Figure 12 is a flow diagram illustrating a method for performing packed multiplication operations on packed data according to one embodiment of the invention.

At step 1201, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for the appropriate multiply operation; SRC1 602, SRC2 603 and DEST 605 addresses in registers 209; signedlunsigned, highllow, and length of the data elements in the packed data.

At step 1202, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provides execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 1130, decoder 202 enables execution unit 130 to perform the appropriate packed multiply operation. Decoder 202 further communicates, via internal bus 170, the size of data elements and the highllow for the multiply operation.

At step 1210, the size of the data element determines which step is to be executed next. If the size of the data elements is eight bits (byte data), then execution unit 130 performs step 1212. However, if the size of the data elements in the packed data is sixteen bits (word data), then execution unit 130 performs step 1214. In one embodiment, only sixteen bit data element size packed multiplies are supported. In another embodiment, eight bit and sixteen bit data element size packed multiplies are supported. However, in another embodiment, a thirty-two bit data element size packed multiply is also supported.

Assuming the size of the data elements is eight bits, then step 1212 is executed. In step 1212, the following is performed. Source1 bits seven through zero are multiplied by Source2 bits seven through zero generating Result bits seven through zero. Source1 bits fifteen through eight are multiplied by Source2 bits fifteen through eight generating Result bits fifteen through eight. Source1 bits twenty-three through sixteen are multiplied by Source2 bits twenty-three through sixteen generating Result bits twenty-three through sixteen. Source1 bits thirty-one through twenty-four are multiplied by Source2 bits thirty-one through twenty-four generating Result bits thirty-one through twenty-four. Source1 bits thirty-nine through thirty-two are multiplied by Source2 bits thirty-nine through thirty-two generating Result bits thirty-nine through thirty-two. Source1 bits forty-seven through forty are multiplied by Source2 bits forty-seven through

forty generating Result forty-seven through forty. Source1 bits fifty-five through forty-eight are multiplied by Source2 bits fifty-five through forty-eight generating Result bits fifty-five through forty-eight. Source1 bits sixty-three through fifty-six are multiplied by Source2 bits generating Result bits sixty-three through fifty-six.

Assuming the size of the data elements is sixteen bits, then step 1214 is executed. In step 1214, the following is performed. Source1 bits fifteen through zero are multiplied by Source2 bits fifteen through zero generating Result bits fifteen through zero. Source1 bits thirty-one through sixteen are multiplied by Source2 bits thirty-one through sixteen generating Result bits thirty-one through sixteen. Source1 bits forty-seven through thirty-two are multiplied by Source2 bits forty-seven through thirty-two generating Result bits forty-seven through thirty-two. Source1 bits sixty-three through forty-eight are multiplied by Source2 bits sixty-three through forty-eight generating Result bits sixty-three through forty-eight.

In one embodiment, the multiplies of step 1212 are performed simultaneously. However, in another embodiment, these multiplies are performed serially. In another embodiment, some of these multiplies are performed simultaneously and some are performed serially. This discussion also applies to the multiplies of step 1214 as well.

At step 1220, the Result is stored in the DEST register.

Table 14 illustrates the in-register representation of packed multiply unsigned high operation on packed word data. The first row of bits is the packed data representation of Source1. The second row of bits is the data representation of Source2. The third row of bits is the packed data representation of the Result. The number below each data element bit is the data element number. For example, Source1 data element two is $11111111\ 00000000_2$.

11111111 11111111	11111111 00000000	11111111 00000000	00001110 00001000
3 Multiply	2 Multiply	1 Multiply	0 Multiply
00000000 00000000	00000000 00000001	10000000 00000000	00001110 10000001
=	=	=	=
00000000 00000000	00000000 00000000	01111111 10000000	00000000 11001011
3	2	1	0

Table 14

Table 15 illustrates the in-register representation of multiply high signed packed operation on packed word data.

11111111 11111111	11111111 00000000	11111111 00000000	00001110 00001000
3 Multiply	2 Multiply	1 Multiply	0 Multiply
00000000 00000000	00000000 00000001	10000000 00000000	00001110 10000001
=	=	=	=
00000000 00000000	11111111 11111111	00000000 10000000	00000000 11001011
3	2	1	0

Table 15

Table 16 illustrates the in-register representation of packed multiply low operation on packed word data.

11111111 11111111	11111111 00000000	11111111 00000000	00001110 00001000
3 Multiply	2 Multiply	1 Multiply	0 Multiply
00000000 00000000	00000000 00000001	10000000 00000000	00001110 10000001
=	=	=	=
00000000 00000000	11111111 00000000	00000000 00000000	10000010 00001000
3	2	1	0

Table 16

PACKED DATA MULTIPLY CIRCUITS

In one embodiment, the multiply operation can occur on multiple data elements in the same number of clock cycles as a single multiply operation on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used. That is, registers are simultaneously instructed to perform the multiply operation on the data elements. This is discussed in more detail below.

Figure 13 illustrates a circuit for performing packed multiplication according to one embodiment of the invention. Operation control 1300 controls the circuits performing the multiplication. Operation control 1300 processes the control signal for the multiply operation and has the following outputs: highllow enable 1380; bytelword enable 1381 and sign enable 1382. Highllow enable 1380 identifies whether the high or low order bits of the product are to be included in the result. Bytelword enable 1381 identifies whether a byte packed data or word packed data multiply operation is to be performed. Sign enable 1382 indicates whether signed multiplication should be used.

Packed word multiplier 1301 multiplies four word data elements simultaneously. Packed byte multiplier 1302 multiplies eight byte data elements. Packed word multiplier 1301 and packed byte multiplier 1302 both have the following inputs: Source1[63:0] 1331, Source2[63:0] 1333, sign enable 1382, and highllow enable 1380.

Packed word multiplier 1301 includes four 16x16 multiplier circuits: 16x16 multiplier A 1310, 16x16 multiplier B 1311, 16x16 multiplier C 1312 and 16x16 multiplier D 1313. 16x16 multiplier A 1310 has as inputs Source1[15:0] and Source2[15:0]. 16x16 multiplier B 1311 has as inputs Source1[31:16] and Source2[31:16]. 16x16 multiplier C 1312 has as inputs Source1[47:32] and Source2[47:32]. 16x16 multiplier D 1313 has as inputs Source1[63:48] and Source2[63:48]. Each 16x16 multiplier is coupled to the sign enable 1382. Each 16x16 multiplier produces a thirty-two bit product. For each multiplier, a multiplexor (Mx0 1350, Mx1 1351, Mx2 1352 and Mx3 1353 respectively) receives the thirty-two bit result. Depending on the value of the highllow enable 1380, each multiplexor outputs the sixteen high order bits or the sixteen low order bits of the product. The outputs of the four multiplexors are combined into one sixty-four bit result. This result is optionally stored in a result register 1 1371.

Packed byte multiplier 1302 includes eight 8x8 multiplier circuits: 8x8 multiplier A 1320 through 8x8 multiplier H 1327. Each 8x8 multiplier has an eight bit input from each of Source1[63:0] 1331 and Source2[63:0] 1333. For example 8x8 multiplier A 1320 has as inputs Source1[7:0] and Source2[7:0] while 8x8 multiplier H 1327 has as inputs Source1[63:56] and Source2[63:56]. Each 8x8 multiplier is coupled to the sign enable 1382. Each 8x8 multiplier produces a sixteen bit product. For each multiplier, a multiplexor (e.g. Mx4 1360 and Mx11 1367) receives the sixteen bit result. Depending on the value of the highllow enable 1380, each multiplexor outputs the eight high order bits or the eight low order bits of the product. The outputs of the eight multiplexors are combined into one sixty-four bit result. This result is optionally stored in a result register 2 1372. The bytelword enable 1381 enables the particular result register, depending on the size of the data element that the operation requires.

In one embodiment, the area used to realize the multiplies is reduced by making circuits that can multiply both two 8x8 numbers or one 16x16 number. That is, two 8x8 multipliers and one 16x16 multiplier are combined into one 8x8 and 16x16 multiplier. Operation control 1300 would enable the appropriate size for the multiply. In such an embodiment, the physical area used by the multipliers would be reduced, however, it would be difficult to execute a packed byte multiply and a packed word multiply. In another embodiment supporting packed doubleword multiplies, one multiplier can perform four 8x8 multiplies, two 16x16 multiplies or one 32x32.

In one embodiment, only a packed word multiply operation is provided. In this embodiment, packed byte multiplier 1302 and result register 2 1372 would not be included.

ADVANTAGES OF INCLUDING THE DESCRIBED PACKED MULTIPLY OPERATION IN THE INSTRUCTION SET

Thus, the described packed multiply instruction provides for the independent multiplication of each data element in Source1 by its respective data element in Source 2. Of course, algorithms that require each element in Source1 to be multiplied by the same number can be performed by storing the same number in each element of Source2. In addition, this multiply instruction insures

against overflows by breaking the carry chains; thereby releasing the programmer of this responsibility, removing the need for instructions to prepare data to prevent overflows, and resulting in more robust code.

In contrast, prior art general purpose processors that do not support such an instruction are required to perform this operation by unpacking the data elements, performing the multiplies, and then packing the results for further packed processing. Thus, processor 109 can multiply different data elements of a packed data by different multipliers in parallel using one instruction.

Typical multimedia algorithms perform a large number of multiply operations. Thus, by reducing the number of instructions required to perform these multiply operations, performance of these multimedia algorithms is increased. Thus, by providing this multiply instruction in the instruction set supported by processor 109, processor 109 can execute algorithms requiring this functionality at a higher performance level.

MULTIPLY-ADD/SUBTRACT

MULTIPLY-ADD/SUBTRACT OPERATIONS

In one embodiment, two multiply-add operations are performed using a single multiply-add instruction as shown below in Table 17a and Table 17b -- Table 17a shows a simplified representation of the disclosed multiply-add instruction, while Table 17b shows a bit level example of the disclosed multiply-add instruction.

Multiply-Add Source1, Source2

A_1	A_2	A_3	A_4	Source1
B_1	B_2	B_3	B_4	Source2
		=		
$A_1B_1+A_2B_2$		$A_3B_3+A_4B_4$		Result1

Table 17a

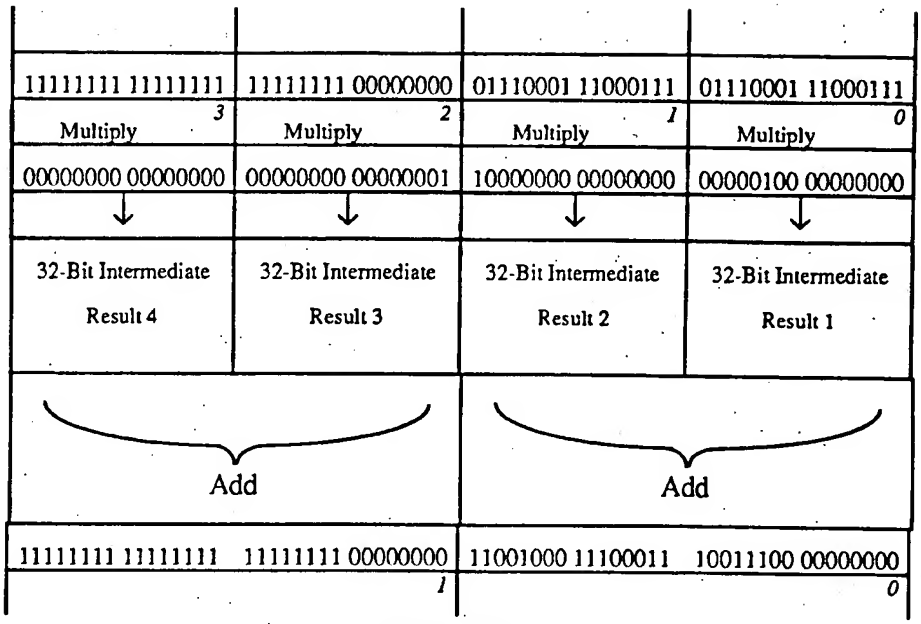


Table 17b

The multiply-subtract operation is the same as the multiply-add operation, except that the add is replaced with a subtract. The operation of an example multiply-subtract instruction which performs two multiply-subtract operations is shown below in Table 12.

Multiply-Subtract Source1, Source2

A_1	A_2	A_3	A_4	Source1
B_1	B_2	B_3	B_4	Source2
=				
A_1B_1-A_2B_2		A_3B_3-A_4B_4		Result1

Table 12

In one embodiment of the invention, the SRC1 register contains packed data (Source1), the SRC2 register contains packed data (Source2), and the DEST register will contain the result (Result) of performing the multiply-add or multiply-subtract instruction on Source1 and Source2. In the first step of the multiply-add or multiply-subtract instruction, Source1 will have each data element independently multiplied by the respective data element of Source2 to

generate a set of respective intermediate results. When executing the multiply-add instruction, these intermediate results are summed by pairs producing two resulting data elements that are stored as data elements of the Result. In contrast, when executing the multiply-subtract instruction, these intermediate results are subtracted by pairs producing two resulting data elements that are stored as data elements of the Result.

Alternative embodiments may vary the number of bits in the data elements, in the intermediate results, and/or in the data elements in the Result. In addition, alternative embodiment may vary the number of data elements in Source1, Source 2, and the Result. For example, if Source1 and Source 2 each have 8 data elements, the multiply-add/subtract instructions may be implemented to produce a Result with 4 data elements (each data element in the Result representing the addition of two intermediate results), 2 data elements (each data element in the result representing the addition of four intermediate results), etc.

Figure 14 is a flow diagram illustrating a method for performing multiply-add and multiply-subtract operations on packed data according to one embodiment of the invention.

At step 1401, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for a multiply-add or multiply-subtract instruction.

At step 1402, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provide execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 1403, decoder 202 enables execution unit 130 to perform the instruction. If the instruction is a multiply-add instruction, flow passes to step 1414. However, if the instruction is a multiply-subtract instruction, flow passes to step 1415.

In step 1414, the following is performed. Source1 bits fifteen through zero are multiplied by Source2 bits fifteen through zero generating a first 32-bit intermediate result (Intermediate Result 1). Source1 bits thirty-one through sixteen are multiplied by Source2 bits thirty-one through sixteen generating a

second 32-bit intermediate result (Intermediate Result 2). Source1 bits forty-seven through thirty-two are multiplied by Source2 bits forty-seven through thirty-two generating a third 32-bit intermediate result (Intermediate Result 3). Source1 bits sixty-three through forty-eight are multiplied by Source2 bits sixty-three through forty-eight generating a fourth 32-bit intermediate result (Intermediate Result 4). Intermediate Result 1 is added to Intermediate Result 2 generating bits thirty-one through 0 of the Result, and Intermediate Result 3 is added to Intermediate Result 4 generating bits sixty-three through thirty-two of the Result.

Step 1415 is the same as step 1414, with the exception that Intermediate Result 1 and Intermediate Result 2 are subtracted to generate bits thirty-one through 0 of the Result, Result 3 and Intermediate Result 4 are subtracted to generate bits sixty-three through thirty-two of the Result.

Different embodiments may be perform the multiplies and adds/subtracts serially, in parallel, or in some combination of serial and parallel operations.

At step 1420, the Result is stored in the DEST register.

PACKED DATA MULTIPLY-ADD/SUBTRACT CIRCUITS

In one embodiment, each of the multiply-add and multiply-subtract instructions can occur on multiple data elements in the same number of clock cycles as a single multiply on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used. That is, registers are simultaneously instructed to perform the multiply-add or multiply-subtract operations on the data elements. This is discussed in more detail below.

Figure 15 illustrates a circuit for performing multiply-add and/or multiply-subtract operations on packed data according to one embodiment of the invention. Operation control 1500 processes the control signal for the multiply-add and multiply-subtract instructions. Operation control 1500 outputs signals on Enable 1580 to control Packed Multiply-Adder/Subtractor 1501.

Packed Multiply-Adder/Subtractor 1501 has the following inputs: Source1[63:0] 1531, Source2[63:0] 1533, and Enable 1580. Packed Multiply-Adder/Subtractor 1501 includes four 16x16 multiplier circuits: 16x16 multiplier A 1510, 16x16 multiplier B 1511, 16x16 multiplier C 1512 and 16x16 multiplier D 1513. 16x16 multiplier A 1510 has as inputs Source1[15:0] and

Source2[15:0]. 16x16 multiplier B 1511 has as inputs Source1[31:16] and Source2[31:16]. 16x16 multiplier C 1512 has as inputs Source1[47:32] and Source2[47:32]. 16x16 multiplier D 1513 has as inputs Source1[63:48] and Source2[63:48]. The 32-bit intermediate results generated by 16x16 multiplier A 1510 and 16x16 multiplier B 1511 are received by Virtual Adder/Subtractor 1550, while the 32-bit intermediate results generated by 16x16 multiplier C 1512 and 16x16 multiplier D 1513 are received by Virtual Adder/Subtractor 1551.

Based on whether the current instruction is a multiply-add or multiply-subtract instruction, Virtual Adder/Subtractor 1550 and Virtual Adder/Subtractor 1551 either add or subtract their respective 32-bit inputs. The output of Virtual Adder/Subtractor 1550 (i.e., bits thirty one through zero of the Result) and the output of Virtual Adder/Subtractor 1551 (i.e., bits 63 through thirty two of the Result) are combined into the 64-bit Result and communicated to Result Register 1571.

In one embodiment, Virtual Adder/Subtractor 1551 and Virtual Adder/Subtractor 1550 are implemented in a similar fashion as Virtual Adder/Subtractor 1108b and Virtual Adder/Subtractor 1108a (i.e., each of Virtual Adder/Subtractor 1551 and Virtual Adder/Subtractor 1550 are composed of four 8-bit adders with the appropriate propagation delays). However, alternative embodiments could implement Virtual Adder/Subtractor 1551 and Virtual Adder/Subtractor 1550 in any number of ways.

To perform the equivalent of these multiply-add or multiply-subtract instructions on prior art processors which operate on unpacked data, four separate 64-bit multiply operations and two 64-bit add or subtract operations, as well as the necessary load and store operations, would be needed. This wastes data lines and circuitry that are used for the bits that are higher than bit sixteen for Source1 and Source 2, and higher than bit thirty two for the Result. As well, the entire 64-bit result generated by such prior art processors may not be of use to the programmer. Therefore, the programmer would have to truncate each result.

ADVANTAGES OF INCLUDING THE DESCRIBED MULTIPLY-ADD OPERATION IN THE INSTRUCTION SET

The described multiply-add/subtract instructions can be used for a number of purposes. For example, the multiply-add instruction can be used for the multiplication of complex numbers and for the multiplication and accumulation of values. Several algorithms which utilize the multiply-add instruction are later described herein.

Thus, by including the described multiply-add and/or multiply-subtract instructions in the instruction set supported by processor 109, many functions can be performed in fewer instructions than prior art general purpose processors which lack these instructions.

PACKED SHIFT

PACKED SHIFT OPERATION

In one embodiment of the invention, the SRC1 register contains the data (Source1) to be shifted, the SRC2 register contains the data (Source2) representing the shift count, and DEST register will contain the result of the shift (Result). That is, Source1 will have each data element independently shifted by the shift count. In one embodiment, Source2 is interpreted as an unsigned 64 bit scalar. In another embodiment, Source2 is packed data and contains shift counts for each corresponding data element in Source1.

In one embodiment of the invention, both arithmetic shifts and logical shifts are supported. An arithmetic shift, shifts the bits of each data element down by a specified number, and fills the high order bit of each data element with the initial value of the sign bit. A shift count greater than seven for packed byte data, greater than fifteen for packed word data, or greater than thirty-one for packed doubleword, causes the each Result data element to be filled with the initial value of the sign bit. A logical shift can operate by shifting bits up or down. In a shift right logical, the high order bits of each data element are filled with zeroes. A shift left logical causes the least significant bits of each data element to be filled with zeroes.

In one embodiment of the invention, a shift right arithmetic, the shift right logical, and the shift left logical operations are supported for packed bytes and packed words. In another embodiment of the invention, these operations are supported for packed doublewords also.

Figure 16 is a flow diagram illustrating a method for performing a packed shift operation on packed data according to one embodiment of the invention.

At step 1601, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for the appropriate shift operation; SRC1 602, SRC2 603 and DEST 605 addresses in registers 209; saturate/unsaturate (not necessarily needed for shift operations), signed/unsigned (again not necessarily needed), and length of the data elements in the packed data.

At step 1602, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provides execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the scalar shift count stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 1603, decoder 202 enables execution unit 130 to perform the appropriate packed shift operation. Decoder 202 further communicates, via internal bus 170, the size of data elements, the type of shift operation, and the direction of the shift (for logical shifts).

At step 1610, the size of the data element determines which step is to be executed next. If the size of the data elements is eight bits (byte data), then execution unit 130 performs step 1612. However, if the size of the data elements in the packed data is sixteen bits (word data), then execution unit 130 performs step 1614. In one embodiment, only eight bit and sixteen bit data element size packed shifts are supported. However, in another embodiment, a thirty-two bit data element size packed shift is also supported.

Assuming the size of the data elements is eight bits, then step 1612 is executed. In step 1612, the following is performed. Source1 bits seven through zero are shifted by the shift count (Source2 bits sixty-three through zero) generating Result bits seven through zero. Source1 bits fifteen through eight are shifted by the shift count generating Result bits fifteen through eight. Source1

bits twenty-three through sixteen are shifted by the shift count generating Result bits twenty-three through sixteen. Source1 bits thirty-one through twenty-four are shifted by the shift count generating Result bits thirty-one through twenty-four. Source1 bits thirty-nine through thirty-two are shifted by the shift count generating Result bits thirty-nine through thirty-two. Source1 bits forty-seven through forty are shifted by the shift count generating Result forty-seven through forty. Source1 bits fifty-five through forty-eight are shifted by the shift count generating Result bits fifty-five through forty-eight. Source1 bits sixty-three through fifty-six are shifted by the shift count generating Result bits sixty-three through fifty-six.

Assuming the size of the data elements is sixteen bits, then step 1614 is executed. In step 1614, the following is performed. Source1 bits fifteen through zero are shifted by the shift count generating Result bits fifteen through zero. Source1 bits thirty-one through sixteen are shifted by the shift count generating Result bits thirty-one through sixteen. Source1 bits forty-seven through thirty-two are shifted by the shift count generating Result bits forty-seven through thirty-two. Source1 bits sixty-three through forty-eight are shifted by the shift count generating Result bits sixty-three through forty-eight.

In one embodiment, the shifts of step 1612 are performed simultaneously. However, in another embodiment, these shifts are performed serially. In another embodiment, some of these shifts are performed simultaneously and some are performed serially. This discussion applies to the shifts of step 1614 as well.

At step 1620, the Result is stored in the DEST register.

Table 19 illustrates the in-register representation of byte packed shift right arithmetic operation. The first row of bits is the packed data representation of Source1. The second row of bits is the data representation of Source2. The third row of bits is the packed data representation of the Result. The number below each data element bit is the data element number. For example, Source1 data element three is 10000000_2.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
Shift 7	Shift 6	Shift 5	Shift 4	Shift 3	Shift 2	Shift 1	Shift 0
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000100
=	=	=	=	=	=	=	=
00000010	00000101	00000101	11111111	11110000	00000111	11111000	11111000
7	6	5	4	3	2	1	0

Table 19

Table 20 illustrates the in-register representation of packed shift right logical operation on packed byte data.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
Shift 7	Shift 6	Shift 5	Shift 4	Shift 3	Shift 2	Shift 1	Shift 0
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000011
=	=	=	=	=	=	=	=
00000101	00001010	00001010	00011111	00010000	00001110	00010001	00010001
7	6	5	4	3	2	1	0

Table 20

Table 21 illustrates the in-register representation of packed shift left logical operation on packed byte data.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
Shift 7	Shift 6	Shift 5	Shift 4	Shift 3	Shift 2	Shift 1	Shift 0
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000011
=	=	=	=	=	=	=	=
01010000	10101000	10101000	11111000	00000000	10000000	01111000	01000000
7	6	5	4	3	2	1	0

Table 21

PACKED DATA SHIFT CIRCUITS

In one embodiment, the shift operation can occur on multiple data elements in the same number of clock cycles as a single shift operation on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used. That is, registers are simultaneously instructed to perform the shift operation on the data elements. This is discussed in more detail below.

Figure 17 illustrates a circuit for performing a packed shift on individual bytes of packed data according to one embodiment of the invention. Figure 17 illustrates the use of a modified byte slice shift circuit, byte slice stage$_i$ 1799. Each byte slice, except for the most significant data element byte slice, includes a shift unit and bit control. The most significant data element byte slice need only have a shift unit.

Shift unit$_i$ 1711 and shift unit$_{i+1}$ 1771 each allow eight bits from Source1 to be shifted by the shift count. In one embodiment, each shift unit operates like a known eight bit shift circuit. Each shift unit has a Source1 input, a Source2 input, a control input, a next stage signal, a last stage signal, and a result output. Therefore, shift unit$_i$ 1711 has Source1$_i$ 1731 input, Source2[63:0] 1733 input, control$_i$ 1701 input, next stage$_i$ 1713 signal, last stage$_i$ 1712 input, and a result stored in result register$_i$ 1751. Therefore, shift unit$_{i+1}$ 1771 has Source1$_{i+1}$ 1732 input, Source2[63:0] 1733 input, control$_{i+1}$ 1702 input, next stage$_{i+1}$ 1773 signal, last stage$_{i+1}$ 1772 input, and a result stored in result register$_{i+1}$ 1752.

The Source1 input is typically an eight bit portion of Source1. The eight bits represents the smallest type of data element, one packed byte data element. Source2 input represents the shift count. In one embodiment, each shift unit receives the same shift count from Source2[63:0] 1733. Operation control 1700 transmits control signals to enable each shift unit to perform the required shift. The control signals are determined from the type of shift (arithmetic/logical) and the direction of the shift. The next stage signal is received from the bit control for that shift unit. The shift unit will shift the most significant bit out/in on the next stage signal, depending on the direction of the shift (left/right). Similarly, each shift unit will shift the least significant bit out/in on the last stage signal, depending on the direction of the shift (right/left). The last stage signal being received from the bit control unit of the previous stage. The result output

represents the result of the shift operation on the portion of Source1 the shift unit is operating upon.

Bit control$_i$ 1720 is enabled from operation control 1700 via packed data enable$_i$ 1706. Bit control$_i$ 1720 controls next stage$_i$ 1713 and last stage$_{i+1}$ 1772. Assume, for example, shift unit$_i$ 1711 is responsible for the eight least significant bits of Source1, and shift unit$_{i+1}$ 1771 is responsible for the next eight bits of Source1. If a shift on packed bytes is performed, bit control$_i$ 1720 will not allow the least significant bit from shift unit$_{i+1}$ 1771 to be communicated with the most significant bit of shift unit$_i$ 1711. However, a shift on packed words is performed, then bit control$_i$ 1720 will allow the least significant bit from shift unit$_{i+1}$ 1771 to be communicated with the most significant bit of shift unit$_i$ 1711.

For example, in Table 22, a packed byte arithmetic shift right is performed. Assume that shift unit$_{i+1}$ 1771 operates on data element one, and shift unit$_i$ 1711 operates on data element zero. Shift unit$_{i+1}$ 1771 shifts its least significant bit out. However operation control 1700 will cause bit control$_i$ 1720 to stop the propagation of that bit, received from last stage$_{i+1}$ 1721, to next stage$_i$ 1713. Instead, shift unit$_i$ 1711 will fill the high order bits with the sign bit, Source1[7].

...	...	...	...	...	...	00001110	10001000
7 Shift	6 Shift	5 Shift	4 Shift	3 Shift	2 Shift	1 Shift	0 Shift
...	...	...	...	...	...	...	00000001
=	=	=	=	=	=	=	=
...	...	...	...	...	...	00001111	01000100
7	6	5	4	3	2	1	0

Table 22

However, if a packed word arithmetic shift is performed, then the least significant bit of shift unit$_{i+1}$ 1771 will be communicated to the most significant bit of shift unit$_i$ 1711. Table 23 illustrates this result. This communication would be allowed for packed doubleword shifts as well.

...	...	...	00001110 10001000
3	2	1	0
Shift	Shift	Shift	Shift
...	...	...	00000001
=	=	=	=
...	...	...	00000111 01000100
3	2	1	0

Table 23

Each shift unit is optionally coupled to a result register. The result register temporarily stores the result of the shift operation until the complete result, Result[63:0] 1760 can be transmitted to the DEST register.

For a complete sixty-four bit packed shift circuit, eight shift units and seven bit control units are used. Such a circuit can also be used to perform a shift on a sixty-four bit unpacked data, thereby using the same circuit to perform the unpacked shift operation and the packed shift operation.

ADVANTAGES OF INCLUDING THE DESCRIBED SHIFT OPERATION IN THE INSTRUCTION SET

The described packed shift instruction causes each element of Source1 to be shifted by the indicated shift count. By including this instruction in the instruction set, each element of a packed data may be shifted using a single instruction. In contrast, prior art general purpose processors that do not support such an operation must perform numerous instructions to unpack Source1, individually shift each unpacked data element, and then pack the results into a packed data format for further packed processing.

MOVE OPERATION

The move operation transfers data to or from registers 209. In one embodiment, SRC2 603 is the address containing the source data and DEST 605 is the address where the data is to be transferred. In this embodiment, SRC1 602 would not be used. In another embodiment, SRC1 602 is equal to DEST 605.

For the purposes of the explanation of the move operation, a distinction is drawn between a register and a memory location. Registers are found in register file 150 while memory can be, for example, in cache 160, main memory 104, ROM 106, data storage device 107.

The move operation can move data from memory to registers 209, from registers 209 to memory, and from a register in registers 209 to a second register in registers 209. In one embodiment, packed data is stored in different registers than those used to store integer data. In this embodiment, the move operation can move data from integer registers 201 to registers 209. For example, in processor 109, if packed data is stored in registers 209 and integer data is stored in integer registers 201, then a move instruction can be used to move data from integer registers 201 to registers 209, and vice versa.

In one embodiment, when a memory address is indicated for the move, the eight bytes of data at the memory location (the memory location containing the least significant byte) are loaded to a register in registers 209 or stored from that register. When a register in registers 209 is indicated, the contents of that register are moved to or loaded from a second register in registers 209. If the integer registers 201 are sixty-four bits in length, and an integer register is specified, then the eight bytes of data in that integer register are loaded to a register in registers 209 or stored from that register.

In one embodiment, integers are represented as thirty-two bits. When a move operation is performed from registers 209 to integer registers 201, then only the low thirty-two bits of the packed data are moved to the specified integer register. In one embodiment, the high order thirty-two bits are zeroed. Similarly, only the low thirty-two bits of a register in registers 209 are loaded when a move is executed from integer registers 201 to registers 209. In one embodiment, processor 109 supports a thirty-two bit move operation between a register in registers 209 and memory. In another embodiment, a move of only thirty-two bits is performed on the high order thirty-two bits of packed data.

PACK OPERATION

In one embodiment of the invention, the SRC1 602 register contains data (Source1), the SRC2 603 register contains the data (Source2), and DEST 605 register will contain the result data (Result) of the operation. That is, parts of Source1 and parts of Source2 will be packed together to generate Result.

In one embodiment, a pack operation converts packed words (or doublewords) into packed bytes (or words) by packing the low order bytes (or words) of the source packed words (or doublewords) into the bytes (or words) of the Result. In one embodiment, the pack operation converts quad packed words into packed doublewords. This operation can be optionally performed with signed data. Further, this operation can be optionally performed with saturate. In an alternative embodiment, additional pack operations are included which operates on the high order portions of each data element.

Figure 18 is a flow diagram illustrating a method for performing pack operations on packed data according to one embodiment of the invention.

At step 1801, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for the appropriate pack operation; SRC1 602, SRC2 603 and DEST 605 addresses in registers 209; saturate/unsaturate, signed/unsigned, and length of the data elements in the packed data. As mentioned previously, SRC1 602 (or SRC2 603) can be used as DEST 605.

At step 1802, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provides execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 1803, decoder 202 enables execution unit 130 to perform the appropriate pack operation. Decoder 202 further communicates, via internal bus 170, saturate and the size of the data elements in Source1 and Source2. Saturate is optionally used to maximize the value of the data in the result data element. If the value of the data elements in Source1 or Source2 are greater than or less than the range of values that the data elements of Result can represent, then the corresponding result data element is set to its highest or lowest value. For

example, if signed values in the word data elements of Source1 and Source2 are smaller than 0x80 (or 0x8000 for doublewords), then the result byte (or word) data elements are clamped to 0x80 (or 0x8000 for doublewords). If signed values in word data elements of Source1 and Source 2 are greater than 0x7F (or 0x7FFF for doublewords) , then the result byte (or word) data elements are clamped to 0x7F (or 0x7FFF).

At step 1810, the size of the data element determines which step is to be executed next. If the size of the data elements is sixteen bits (packed word 402 data), then execution unit 130 performs step 1812. However, if the size of the data elements in the packed data is thirty-two bits (packed doubleword 403 data), then execution unit 130 performs step 1814.

Assuming the size of the source data elements is sixteen bits, then step 1812 is executed. In step 1812, the following is performed. Source1 bits seven through zero are Result bits seven through zero. Source1 bits twenty-three through sixteen are Result bits fifteen through eight. Source1 bits thirty-nine through thirty-two are Result bits twenty-three through sixteen. Source1 bits sixty-three through fifty-six are Result bits thirty-one through twenty-four. Source2 bits seven through zero are Result bits thirty-nine through thirty-two. Source2 bits twenty-three through sixteen are Result bits forty-seven through forty. Source2 bits thirty-nine through thirty-two are Result bits fifty-five through forty-eight. Source2 bits sixty-three through fifty-six are Result bits thirty-one through twenty-four. If saturate is set, then the high order bits of each word are tested to determine whether the Result data element should be clamped.

Assuming the size of the source data elements is thirty-two bits, then step 1814 is executed. In step 1814, the following is performed: Source1 bits fifteen through zero are Result bits fifteen through zero. Source1 bits forty-seven through thirty-two are Result bits thirty-one through sixteen. Source2 bits fifteen through zero are Result bits forty-seven through thirty-two. Source2 bits forty-seven through thirty-two are Result bits sixty-three through forty-eight. If saturate is set, then the high order bits of each doubleword are tested to determine whether the Result data element should be clamped.

In one embodiment, the packing of step 1812 is performed simultaneously. However, in another embodiment, this packing is performed serially. In another

embodiment, some of the packing is performed simultaneously and some is performed serially. This discussion also applies to the packing of step 1814.

At step 1820, the Result is stored in the DEST 605 register.

Table 24 illustrates the in-register representation of a pack word operation. The subscripted H_S and L_S represent the high and low order bits, respectively, of each 16-bit data element in Source1 and Source2. For example, A_L represents the low order 8 bits of the data element A in Source1.

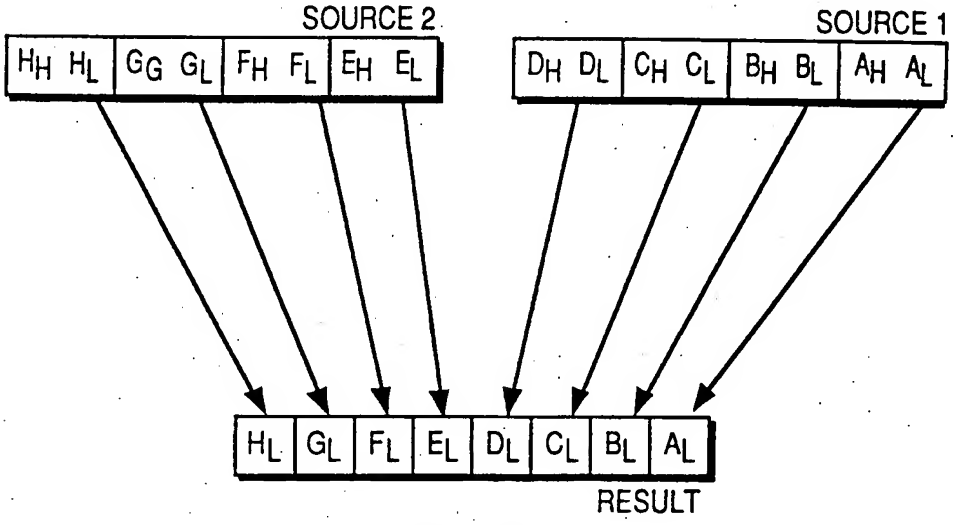


Table 24

Table 25 illustrates the in-register representation of a pack doubleword operation, where the subscripted H_S and L_S represent the high low order bits, respectively, of each 32-bit data element in Source1 and Source2.

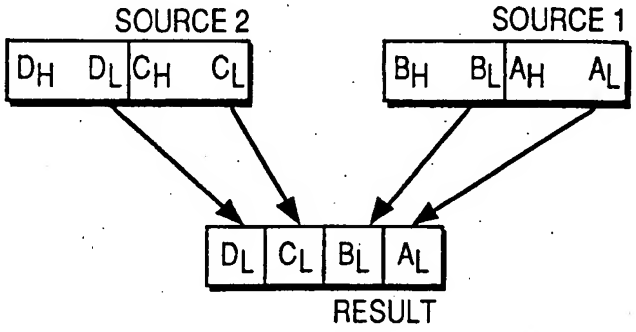


Table 25

PACK CIRCUITS

In one embodiment of the invention, to achieve efficient execution of pack operations parallelism is used. Figures 19a and 19b illustrate a circuit for performing pack operations on packed data according to one embodiment of the invention. The circuit can optionally perform the pack operation with saturation.

The circuit of Figures 19a and 19b includes an operation control 1900, a result register 1952, a result register 1953, eight sixteen bit to eight bit test saturate circuits, and four thirty-two bit to sixteen bit test saturate circuits.

Operation control 1900 receives information from the decoder 202 to enable a pack operation. Operation control 1900 uses the saturate value to enable the saturation tests for each of the test saturate circuits. If the size of the source packed data is word packed data 503, then output enable 1931 is set by operation control 1900. This enables the output of result register 1952. If the size of the source packed data is doubleword packed data 504, then output enable 1932 is set by operation control 1900. This enables the output of output register 1953.

Each test saturate circuit can selectively test for saturation. If a test for saturation is disabled, then each test saturate circuit merely passes the low order bits through to a corresponding position in a result register. If a test for saturate is enabled, then each test saturate circuit tests the high order bits to determine if the result should be clamped.

Test saturate 1910 through test saturate 1917 have sixteen bit inputs and eight bit outputs. The eight bit outputs are the lower eight bits of the inputs, or optionally, are a clamped value (0x80, 0x7F, or 0xFF). Test saturate 1910 receives Source1 bits fifteen through zero and outputs bits seven through zero for result register 1952. Test saturate 1911 receives Source1 bits thirty-one through sixteen and outputs bits fifteen through eight for result register 1952. Test saturate 1912 receives Source1 bits forty-seven through thirty-two and outputs bits twenty-three through sixteen for result register 1952. Test saturate 1913 receives Source1 bits sixty-three through forty-eight and outputs bits thirty-one through twenty-four for result register 1952. Test saturate 1914 receives Source2 bits fifteen through zero and outputs bits thirty-nine through thirty-two for result register 1952. Test saturate 1915 receives Source2 bits thirty-one through sixteen and outputs bits forty-seven through forty for result register 1952. Test saturate 1916 receives Source2 bits forty-seven through thirty-two and outputs bits fifty-

five through forty-eight for result register 1952. Test saturate 1917 receives Source2 bits sixty-three through forty-eight and outputs bits sixty-three through fifty-six for result register 1952.

Test saturate 1920 through test saturate 1923 have thirty-two bit inputs and sixteen bit outputs. The sixteen bit outputs are the lower sixteen bits of the inputs, or optionally, are a clamped value (0x8000, 0x7FFF, or 0xFFFF). Test saturate 1920 receives Source1 bits thirty-one through zero and outputs bits fifteen through zero for result register 1953. Test saturate 1921 receives Source1 bits sixty-three through thirty-two and outputs bits thirty-one through sixteen for result register 1953. Test saturate 1922 receives Source2 bits thirty-one through zero and outputs bits forty-seven through thirty-two for result register 1953. Test saturate 1923 receives Source2 bits sixty-three through thirty-two and outputs bits sixty-three though forty-eight of result register 1953.

For example, in Table 26, a pack word unsigned with no saturate is performed. Operation control 1900 will enable result register 1952 to output result[63:0] 1960.

Source1							
...		...		00001110	01110000	00001110	00001000
3		2		1		0	
Source2							
				00001110	10000001	00001110	10000001
3		2		1		0	
Result							
...	...	10000001	10000001	...	...	01110000	00001000
7	6	5	4	3	2	1	0

Table 26

However, if a pack doubleword unsigned with no saturate is performed, operation control 1900 will enable result register 1953 to output result[63:0] 1960. Table 27 illustrates this result.

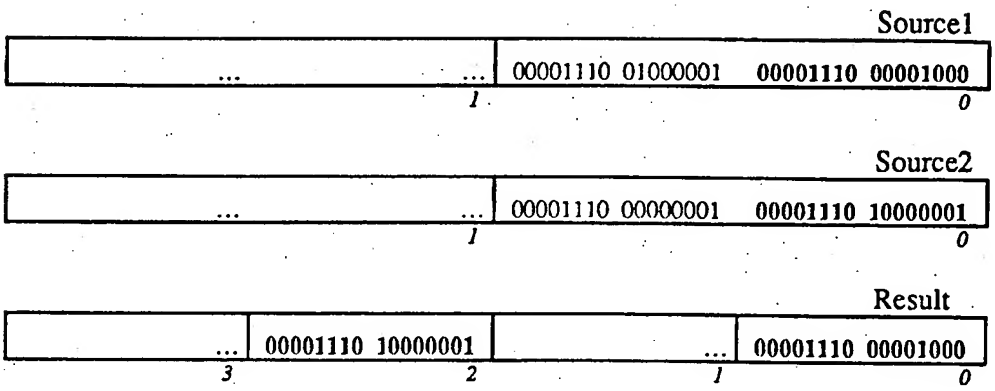


Table 27

ADVANTAGES OF INCLUDING THE DESCRIBED PACK OPERATION IN THE INSTRUCTION SET

The described pack instruction packs a predefined number of bits from each data element in Source1 and Source 2 to generate the Result. In this manner, processor 109 can pack data in as little as half the instructions required by prior art general purpose processors. For example, generating a result which contains four 16-bit data elements from four 32-bit data elements requires only one instruction (as opposed to 2 instructions) as shown below:

Pack.High Source1,Source2

A0.	.A0	C0.	.C0	Source1
G0.	.G0	B0.	.B0	Source2
		=		
A0.	C0.	G0.	B0.	Result1

Table 28

Typical multimedia applications pack large amounts of data. Thus, by reducing the number of instructions required to pack this data by as much as half, performance of these multimedia applications is increased.

UNPACK OPERATION

UNPACK OPERATION

In one embodiment, an unpack operation interleaves the low order packed bytes, words or doublewords of two source packed data to generate result packed bytes, words, or doublewords. This operation is referred to herein as an unpack low operation. In another embodiment, an unpack operation could also interleave the high order elements (referred to as the unpack high operation).

Figure 20 is a flow diagram illustrating a method for performing unpack operations on packed data according to one embodiment of the invention.

Step 2001 and step 2002 are executed first. At step 2003, decoder 202 enables execution unit 130 to perform the unpack operation. Decoder 202 communicates, via internal bus 170, the size of the data elements in Source1 and Source2.

At step 2010, the size of the data element determines which step is to be executed next. If the size of the data elements is eight bits (packed byte 401 data), then execution unit 130 performs step 2012. However, if the size of the data elements in the packed data is sixteen bits (packed word 402 data), then execution unit 130 performs step 2014. However, if the size of the data elements in the packed data is thirty-two bits (packed doubled word 503 data), then execution unit 130 performs step 2016.

Assuming the size of the source data elements is eight bits, then step 2012 is executed. In step 2012, the following is performed. Source1 bits seven through zero are Result bits seven through zero. Source2 bits seven through zero are Result bits fifteen through eight. Source1 bits fifteen through eight are Result bits twenty-three through sixteen. Source2 bits fifteen through eight are Result bits thirty-one through twenty-four. Source1 bits twenty-three through sixteen are Result bits thirty-nine through thirty-two. Source2 bits twenty-three through sixteen are Result bits forty-seven through forty. Source1 bits thirty-one through twenty-four are Result bits fifty-five through forty-eight. Source2 bits thirty-one through twenty-four are Result bits sixty-three through fifty-six.

Assuming the size of the source data elements is sixteen bits, then step 2014 is executed. In step 2014, the following is performed. Source1 bits fifteen through zero are Result bits fifteen through zero. Source2 bits fifteen through zero are Result bits thirty-one through sixteen. Source1 bits thirty-one through

sixteen are Result bits forty-seven through thirty-two. Source2 bits thirty-one through sixteen are Result bits sixty-three through forty-eight.

Assuming the size of the source data elements is thirty-two bits, then step 2016 is executed. In step 2016, the following is performed. Source1 bits thirty-one through zero are Result bits thirty-one through zero. Source2 bits thirty-one through zero are Result bits sixty-three through thirty-two.

In one embodiment, the unpacking of step 2012 is performed simultaneously. However, in another embodiment, this unpacking is performed serially. In another embodiment, some of the unpacking is performed simultaneously and some is performed serially. This discussion also applies to the unpacking of step 2014 and step 2016.

At step 2020, the Result is stored in the DEST 605 register.

Table 29 illustrates the in-register representation of an unpack doubleword operation (each of data elements A_{0-1} and B_{0-1} contain 32 bits).

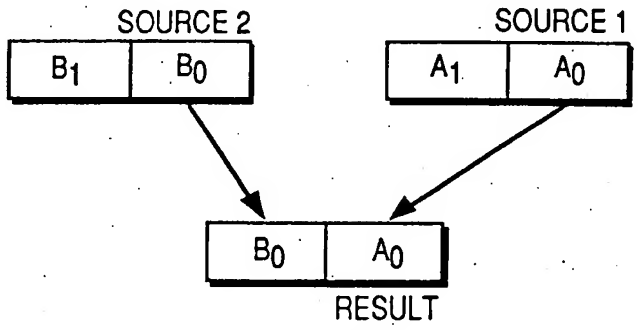


Table 29

Table 30 illustrates the in-register representation of an unpack word operation (each of data elements A_{0-3} and B_{0-3} contain 16 bits).

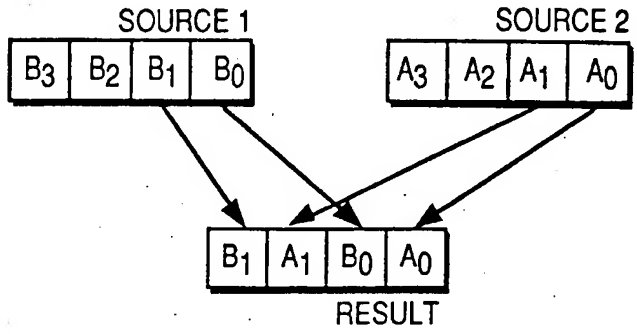


Table 30

Table 31 illustrates the in-register representation of an unpack byte operation (each of data elements A_{0-7} and B_{0-7} contain 8 bits).

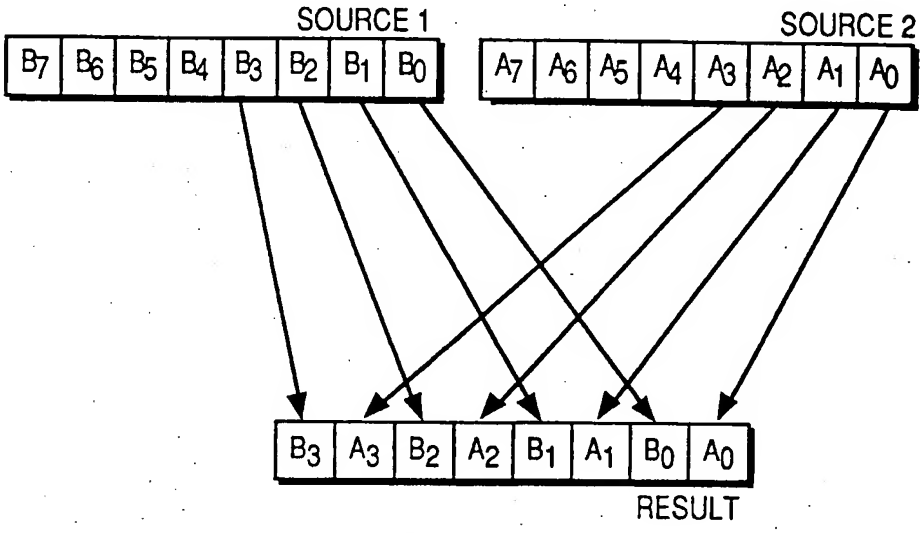


Table 31

UNPACK CIRCUITS

Figure 21 illustrates a circuit for performing unpack operations on packed data according to one embodiment of the invention. The circuit of Figure 21 includes the operation control circuit 2100, a result register 2152, a result register 2153, and a result register 2154.

Operation control 2100 receives information from the decoder 202 to enable an unpack operation. If the size of the source packed data is byte packed data 502, then output enable 2132 is set by operation control 2100. This enables the output of result register 2152. If the size of the source packed data is word packed data 503, then output enable 2133 is set by operation control 2100. This enables the output of output register 2153. If the size of the source packed data is doubleword packed data 504, then output enable 2134 is set by operation control 2100. This enables the output of output result register 2154.

Result register 2152 has the following inputs. Source1 bits seven through zero are bits seven through zero for result register 2152. Source2 bits seven through zero are bits fifteen through eight for result register 2152. Source1 bits fifteen through eight are bits twenty-three through sixteen for result register 2152. Source 2 bits fifteen through eight are bits thirty-one through twenty-four for result register 2152. Source1 bits twenty-three through sixteen are bits thirty-nine through thirty-two for result register 2152. Source2 bits twenty-three through sixteen are bits forty-seven through forty for result register 2152. Source1 bits thirty-one through twenty-four are bits fifty-five through forty-eight for result register 2152. Source2 bits thirty-one through twenty-four are bits sixty-three through fifty-six for result register 2152.

Result register 2153 has the following inputs. Source1 bits fifteen through zero are bits fifteen through zero for result register 2153. Source2 bits fifteen through zero are bits thirty-one through sixteen for result register 2153. Source1 bits thirty-one through sixteen are bits forty-seven through thirty-two for result register 2153. Source2 bits thirty-one through sixteen are bits sixty-three though forty-eight of result register 1953.

Result register 2154 has the following inputs. Source1 bits thirty-one through zero are bits thirty-one through zero for result register 2154. Source2 bits thirty-one through zero are bits sixty-three through thirty-two of result register 2154.

For example, in Table 32, an unpack word operation is performed. Operation control 2100 will enable result register 2153 to output result[63:0] 2160.

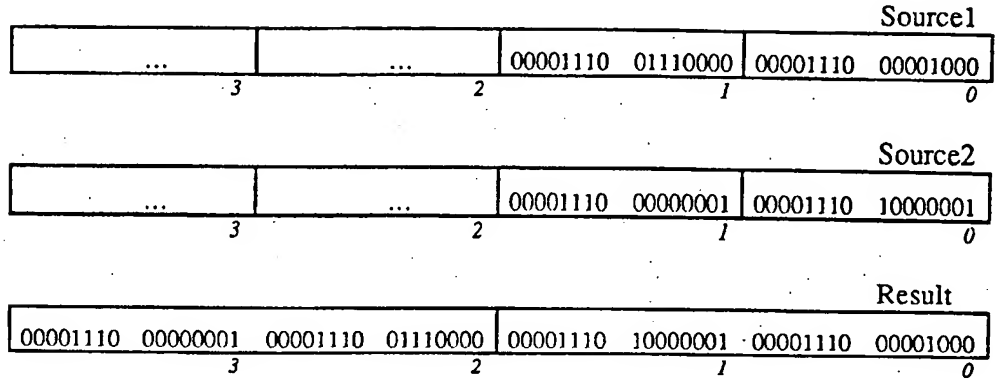


Table 32

However, if an unpack doubleword is performed, operation control 2100 will enable result register 2154 to output result[63:0] 2160. Table 33 illustrates this result.

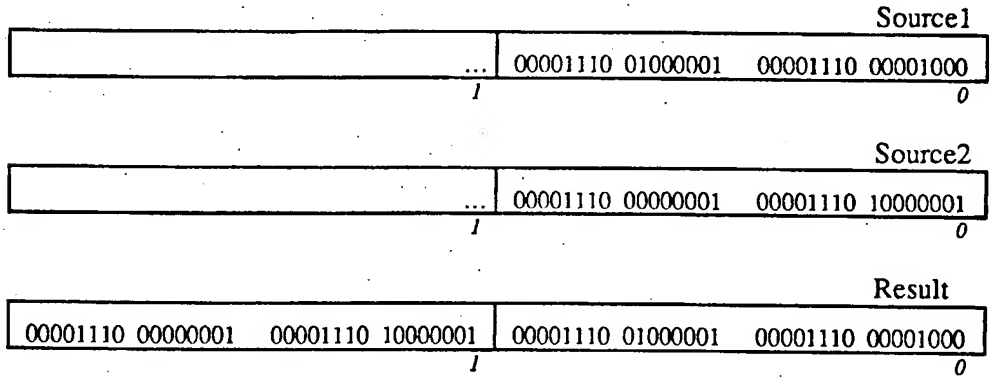


Table 33

ADVANTAGES OF INCLUDING THE DESCRIBED UNPACK INSTRUCTION IN THE INSTRUCTION SET

By including the described unpack instruction in the instruction set, packed data may be either interleaved or unpacked. This unpack instruction can be used for unpacking packed data by making all of the data elements in Source2 all 0s. An example of unpacking bytes is shown below in Table 34a.

Source1

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
7	6	5	4	3	2	1	0

Source2

00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
7	6	5	4	3	2	1	0

Result

00000000 10000000	00000000 01110000	00000000 10001111	00000000 10001000
3	2	1	0

Table 34a

This same unpack instruction can be used for interleaving data as shown in Table 34b. Interleaving is useful in a number of multimedia algorithms. For example, interleaving is useful for transposing matrixes and interpolating pixels.

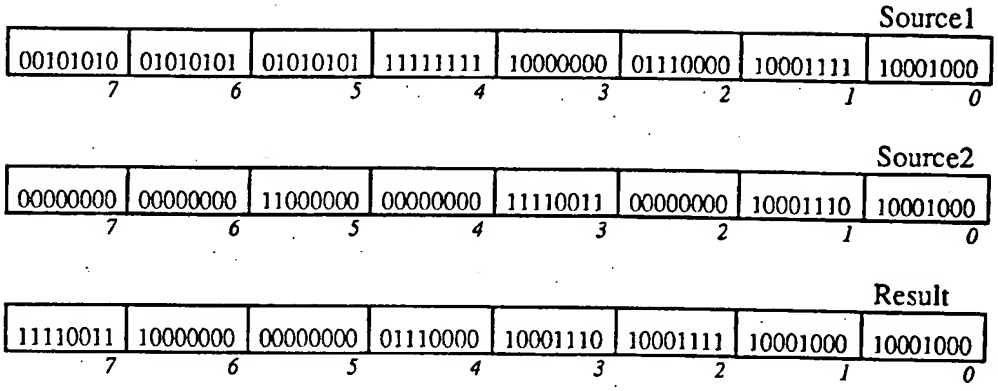


Table 34b

Thus, by providing this unpack instruction in the instruction set supported by processor 109, processor 109 is more versatile and can perform algorithms requiring this functionality at a higher performance level.

POPULATION COUNT

POPULATION COUNT

One embodiment of the invention enables population count operations to be performed on packed data. That is, the invention generates a result data element for each data element of a first packed data. Each result data element represents the number of bits set in each corresponding data element of the first packed data. In one embodiment, the total number of bits set to one is counted.

Table 35a illustrates an in-register representation of a population count operation on a packed data. The first row of bits is the packed data representation of a Source1 packed data. The second row of bits is the packed data representation of the Result packed data. The number below each data element bit is the data element number. For example, Source1 data element 0 is 1000111110001000_2. Therefore, if the data elements are sixteen bits in length (word data), and a population count operation is performed, Execution unit 130 produces the Result packed data as shown.

01110010 00000101	11111111 11111111	01111111 11111111	10001111 10001000
= 3	= 2	= 1	= 0
00000000 00000110	00000000 00010000	00000000 00001111	00000000 00000111
3	2	1	0

Table 35a

In another embodiment, population counts are performed on eight bit data elements. Table 35b illustrates an in-register representation of a population count on a packed data having eight eight-bit packed data elements.

01111111	01010101	10101010	10000001	10000000	11111111	11001111	00000000
= 7	= 6	= 5	= 4	= 3	= 2	= 1	= 0
00000111	00000100	00000100	00000010	00000001	00001000	00000110	00000000
7	6	5	4	3	2	1	0

Table 35b

In another embodiment, population counts are performed on thirty-two bit data elements. Table 35c illustrates an in-register representation of a population count on a packed data having two, thirty-two bit, packed data elements.

11111111 11111111 11111111 11111111	10000000 11110000 11001111 10001000
= 1	= 0
00000000 00000000 00000000 00100000	00000000 00000000 00000000 00001101
1	0

Table 35c

Population counts can also be performed on sixty-four bit integer data. That is, the number of bits set to one, in sixty-four bits of data, is totaled. Table 35d illustrates an in-register representation of a population count on sixty-four bit integer data.

11111111 11111111 11111111 11111111 10000000 11110000 11001111 10001000
=
00000000 00000000 00000000 00100000 00000000 00000000 00000000 00101101

Table 35d

A METHOD OF PERFORMING A POPULATION COUNT

Figure 22 is a flow diagram illustrating a method for performing a population count operation on packed data according to one embodiment of the invention. At step 2201, responsive to receiving a control signal 207, decoder 202 decodes that control signal 207. In one embodiment, control signal 207 is supplied via bus 101. In another embodiment, control signal 207 is supplied by cache 160. Thus, decoder 202 decodes: the operation code for population count, and SRC1 602 and DEST 605 addresses in registers 209. Note that SRC2 603 is not used in this present embodiment of the invention. As well, saturate/unsaturate, signed/unsigned, and length of the data elements in the packed data are not used in this embodiment. In the present embodiment of the invention, only sixteen bit data element length packed addition is supported. However, one skilled in the art would understand that population counts can be performed on packed data having eight packed byte data elements or two packed doubleword data elements.

At step 2202, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 address. Registers 209 provides Execution unit 130 with the packed data, Source1, stored in the register at this address. That is, registers 209 communicate the packed data to Execution unit 130 via internal bus 170.

At step 2130, decoder 202 enables Execution unit 130 to perform a population count operation. In an alternative embodiment, decoder 202 further communicates, via internal bus 170, the length of packed data elements.

At step 2205, assuming the length of the data elements is sixteen bits, then Execution unit 130 totals the number of bits set in bit fifteen through bit zero of Source1, producing bit fifteen through bit zero of Result packed data. In parallel

with this totaling, Execution unit 130 adds totals thirty-one through bit sixteen of Source1, producing bit thirty-one through bit sixteen of Result packed data. In parallel with the generation of these totals, Execution unit 130 totals bit forty-seven through bit thirty-two of Source1, producing bit forty-seven through bit thirty-two of Result packed data. In parallel with the generation of these totals, Execution unit 130 totals bit sixty-three through bit forty-eight of Source1, producing bit sixty-three through bit forty-eight of Result packed data.

At step 2206, decoder 202 enables a register in registers 209 with DEST 605 address of the destination register. Thus, the Result packed data is stored in the register addressed by DEST 605.

A METHOD OF PERFORMING A POPULATION COUNT ON ONE DATA ELEMENT

Figure 23 is a flow diagram illustrating a method for performing a population count operation on one data element of a packed data and generating a single result data element for a result packed data according to one embodiment of the invention. At step 2310a, a column sum, CSum1a, and a column carry, CCarry 1a, are generated from Source1 bits fifteen, fourteen, thirteen and twelve. At step 2310b, a column sum, CSum1b, and a column carry, CCarry 1b, are generated from Source1 bits eleven, ten, nine and eight. At step 2310c, a column sum, CSum1c, and a column carry, CCarry 1c, are generated from Source1 bits seven, six, five and four. At step 2310d, a column sum, CSum1d, and a column carry, CCarry 1d, are generated from Source1 bits three, two, one and zero. In one embodiment of the invention, steps 2310a-d are performed in parallel. At step 2320a, a column sum, CSum2a, and a column carry, CCarry 2b, are generated from CSum1a, CCarry1a, CSum1b, and CCarry1b. At step 2320b, a column sum, CSum2b, and a column carry, CCarry 2b, are generated from CSum1c, CCarry1, CSum1d, and CCarry1d. In one embodiment of the invention, steps 2320a-b are performed in parallel. At step 2330, a column sum, CSum3, and a column carry, CCarry 3, are generated from CSum2a, CCarry2a, CSum2b, and CCarry2b. At step 2340, a Result is generated from CSum3 and CCarry3. In one embodiment, the Result is represented in sixteen bits. In this embodiment, as only bit four through bit zero are need to represent the maximum number of bits set in a Source1, bits fifteen through five are set to zero. The maximum number of bits for Source1 is sixteen. This occurs when Source1

equals 1111111111111111_2. The Result would be sixteen and would be represented by 0000000000010000_2.

Thus, to calculate four result data elements for a population count operation on a sixty-four bit packed data, the steps of Figure 23 would be performed for each data element in the packed data. In one embodiment, the four sixteen bit result data elements would be calculated in parallel.

A CIRCUIT FOR PERFORMING A POPULATION COUNT

Figure 24 illustrates a circuit for performing a population count operation on packed data having four word data elements according to one embodiment of the invention. Figure 25 illustrates a detailed circuit for performing a population count operation on one word data element of a packed data according to one embodiment of the invention.

Figure 24 illustrates a circuit wherein Source1 bus 2401 carries information signals to the popcnt circuits 2408a-d via $Source1_{IN}$ 2406a-d. Thus, popcnt circuit 2408a totals the number of bits set in bit fifteen through bit zero of Source1, producing bit fifteen through bit zero of Result. Popcnt circuit 2408b totals the number of bits set in bit thirty-one through bit sixteen of Source1, producing bit thirty-one through bit sixteen of Result. Popcnt circuit 2408c totals the number of bits set in bit forty-seven through bit thirty-two of Source1, producing bit forty-seven through bit thirty-two of Result. Popcnt circuit 2408d totals the number of bits set in bit sixty-three through bit forty-eight of Source1, producing bit sixty-three through bit forty-eight of Result. Enable 2404a-d receives, from Operation Control 2410, via control 2403, control signals enabling popcnt circuits 2408a-d to perform population count operations, and to place a Result on the Result Bus 2409. One skilled in the art would be able to create such a circuit given the above description and the above description and illustrations in Figures 1-6b and 23-25.

Popcnt circuits 2408a-d communicate result information of a packed population count operation onto Result bus 2409, via result out 2407a-d. This result information is then stored in the integer register specified by the DEST 605 register address.

A CIRCUIT FOR PERFORMING A POPULATION COUNT ON ONE DATA ELEMENT

Figure 25 illustrates a detailed circuit for performing a population count operation on one, word, data element of a packed data. In particular, Figure 25 illustrates a portion of popcnt circuit 2408a. To achieve the maximum performance for applications employing a population count operation, the operation should be complete within one clock cycle. Therefore, given that accessing a register and storing a result requires a certain percentage of the clock cycle, the circuit of Figure 24 completes its operation within approximately 80% of one clock period. This circuit has the advantage of allowing processor 109 to execute a population count operation on four sixteen bit data elements in one clock cycle.

Popcnt circuit 2408a employs 4->2 carry-save adders (unless otherwise specified, CSA will refer to a 4->2 carry-save adder). 4->2 carry-save adders, as may be employed in the popcnt circuit 2408a-d, are well known in the art. A 4->2 carry-save adder is an adder that adds four operands, resulting in two sums. Since the population count operation in popcnt circuit 2408a involves sixteen bits, the first level includes four 4->2 carry-save adders. These four 4->2 carry-save adders transform the sixteen one-bit operands into eight two-bit sums. The second level transforms the eight two-bit sums into four three-bit sums, and the third level transforms the four three-bit sums into two four-bit sums. Then a four-bit full adder, adds the two four-bit sums to generate a final result.

Although 4->2 carry-save adders are used, an alternative embodiments could employ 3->2 carry-save adders. Alternatively, a number of full adders could be used; however, this configuration would not provide a result as quickly as the embodiment shown in Figure 25.

$Source1_{IN\ 15\text{-}0}$ 2406a carries bit fifteen through bit zero of Source1. The first four bits are coupled to the inputs of a 4->2 carry-save adder (CSA 2510a). The next four bits are coupled to the inputs of CSA 2510b. The next four bits are coupled to the inputs of CSA 2510c. The final four bits are coupled to the inputs of CSA 2510d. Each CSA 2510a-d generates two, two-bit, outputs. The two, two bit, outputs of CSA 2510a are coupled to two inputs of CSA 2520a. The two, two bit, outputs of CSA 2510b are coupled to the other two inputs of CSA 2520a. The two, two bit outputs of CSA 2510c are coupled to two inputs of CSA 2520b. The two, two bit outputs of CSA 2510d are coupled to the other two

inputs of CSA 2520b. Each CSA 2520a-b generates two, three bit, outputs. The two, three bit, outputs of 2520a are coupled to two inputs of CSA 2530. The two , three bit, outputs of 2520b are coupled to the other two inputs of CSA 2530. CSA 2530 generates two, four bit, outputs.

These two four bit outputs are coupled to two inputs of a full adder (FA 2550). FA 2550 adds the two four bit inputs and communicates bit three through bit zero of Result Out 2407a as a total of the addition of the two, four bit, inputs. FA 2550 generates bit four of Result Out 2407a through carry out (CO 2552). In an alternative embodiment, a five bit full adder is used to generate bit four through bit zero of Result Out 2407a. In either case, bit fifteen through bit five of Result Out 2407a are tied to zero. As well, any carry inputs to the full adder are tied to zero.

Although not shown in Figure 25, one skilled in the art would understand that Result Out 2407a could be multiplexed or buffered onto Result bus 2409. The multiplexor would be controlled by Enable 2404a. This would allow other Execution unit circuits to write data onto Result bus 2409.

ADVANTAGES OF INCLUDING THE DESCRIBED POPULATION COUNT OPERATION IN THE INSTRUCTION SET

The described population count instruction calculates the number of bits set in each of the data elements of packed data, such as Source1. Thus, by including this instruction in the instruction set, a population count operation may be performed on packed data in a single instruction. In contrast, prior art general purpose processors must perform numerous instructions to unpack Source1, perform the function individually on each unpacked data element, and then pack the results for further packed processing.

Thus, by providing this population count instruction in the instruction set supported by processor 109, the performance of algorithms requiring this functionality is increased.

LOGICAL OPERATIONS

LOGICAL OPERATIONS

In one embodiment of the invention, the SRC1 register contains packed data (Source1), the SRC2 register contains packed data (Source2), and the DEST register will contain the result (Result) of performing the selected logical operation on Source1 and Source2. For example, if the logical AND operation is selected, Source1 will be logically ANDed with Source 2.

In one embodiment of the invention, the following logical operations are supported: logical AND, logical ANDN, logical OR, and logical XOR. The logical AND, OR, and XOR operations are well known in the art. The logical ANDN operation causes Source2 to be ANDed with the logical inversion of Source 1. While the invention is described in relation to these logical operations, alternative embodiments could implement other logical operations.

Figure 26 is a flow diagram illustrating a method for performing a number of logical operations on packed data according to one embodiment of the invention.

At step 2601, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for the appropriate logical operation (i.e., AND, ANDN, OR, or XOR); SRC1 602, SRC2 603 and DEST 605 addresses in registers 209.

At step 2602, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provide execution unit 130 with the packed data stored in the SRC1 602 register (Source1) and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 2603, decoder 202 enables execution unit 130 to perform the selected one of the packed logical operations.

At step 2610, the selected one of the packed logical operations determines which step is to be executed next. Execution unit 130 performs step 2612 if the logical AND operation was selected; Execution unit 130 performs step 2613 if the logical ANDN operation was selected; Execution unit 130 performs step

2614 if the logical OR operation was selected; and Execution unit 130 performs step 2615 if the logical XOR operation was selected.

Assuming the logical AND operation was selected, step 2612 is executed. In step 2612, Source1 bits sixty-three through zero are ANDed with Source2 bits sixty-three through zero to generate Result bits sixty-three through zero.

Assuming the logical ANDN operation was selected, step 2613 is executed. In step 2613, Source1 bits sixty-three through zero are ANDNed with Source2 bits sixty-three through zero to generate Result bits sixty-three through zero.

Assuming the logical OR operation was selected, step 2614 is executed. In step 2614, Source1 bits sixty-three through zero are ORed with Source2 bits sixty-three through zero to generate Result bits sixty-three through zero.

Assuming the logical XOR operation was selected, step 2615 is executed. In step 2615, Source1 bits sixty-three through zero are exclusive ORed with Source2 bits sixty-three through zero to generate Result bits sixty-three through zero.

At step 2620, the Result is stored in the DEST register.

Table 36 illustrates the in-register representation of a logical ANDN operation on packed data. The first row of bits is the packed data representation of Source1. The second row of bits is the packed data representation of Source2. The third row of bits is the packed data representation of the Result. The number below each data element bit is the data element number. For example, Source1 data element two is 11111111 00000000_2.

11111111 11111111	11111111 00000000	11111111 00000000	00001110 00001000
3 Logical ANDN	2 Logical ANDN	1 Logical ANDN	0 Logical ANDN
00000000 00000000	00000000 00000001	10000000 00000000	00001110 10000001
=	=	=	=
00000000 00000000	00000000 00000001	00000000 00000000	00000000 10000001
3	2	1	0

Table 36

While the invention is described in relation to the same logical operation being performed on corresponding data elements in Source1 and Source2, alternative embodiments could support instructions which allowed for the logical

operation performed on corresponding data elements to be selected on a per element basis.

PACKED DATA LOGICAL CIRCUITS

In one embodiment, the described logical operations can occur on multiple data elements in the same number of clock cycles as a single logical operation on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used.

Figure 27 illustrates a circuit for performing logical operations on packed data according to one embodiment of the invention. Operation control 2700 controls the circuits performing the logical operations. Operation control 2700 processes the control signal and outputs selection signals on control lines 2780. These selection signals communicate to Logical Operations Circuit 2701 the selected one of the AND, ANDN, OR, and XOR operations.

Logical Operations Circuit 2701 receives Source1 [63:0] and Source2 [63:0] and performs the logical operation indicated by the selection signals to generate the Result. Logical Operations Circuit 2701 communicates Result [63:0] to Result Register 2731.

ADVANTAGES OF INCLUDING THE DESCRIBED LOGICAL OPERATIONS IN THE INSTRUCTION SET

The described logical instructions perform a logical AND, a logical AND NOT, a logical OR, and a logical OR NOT. These instructions are useful in any application that requires logical manipulation of data. By including these instructions in the instruction set supported by processor 109, these logical operations may be performed on packed data in one instruction.

PACKED COMPARE

PACKED COMPARE OPERATION

In one embodiment of the invention, the SRC1 602 register contains data (Source1) to be compared, the SRC2 603 register contains the data (Source2) to be compared against, and DEST 605 register will contain the result of the compare (Result). That is, Source1 will have each data element independently compared by the each data element of Source2, according to an indicated relationship.

In one embodiment of the invention, the following compare relationships are supported: equal; signed greater than; signed greater than or equal; unsigned greater than; or unsigned greater than or equal. The relationship is tested in each pair of corresponding data elements. For example, Source1[7:0] is greater than Source2[7:0], with the result being Result[7:0]. If the result of the comparison satisfies the relationship, then, in one embodiment, the corresponding data element in Result is set to all ones. If the result of the comparison does not satisfy the relationship, then the corresponding data element in Result is set to all zeroes.

Figure 28 is a flow diagram illustrating a method for performing packed compare operations on packed data according to one embodiment of the invention.

At step 2801, decoder 202 decodes control signal 207 received by processor 109. Thus, decoder 202 decodes: the operation code for the appropriate compare operation; SRC1 602, SRC2 603 and DEST 605 addresses in registers 209; saturate/unsaturate (not necessarily needed for compare operations), signed/unsigned, and length of the data elements in the packed data. As mentioned previously, SRC1 602 (or SRC2 603) can be used as DEST 605.

At step 2802, via internal bus 170, decoder 202 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provides execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

At step 2803, decoder 202 enables execution unit 130 to perform the appropriate packed compare operation. Decoder 202 further communicates, via internal bus 170, the size of data elements and the relationship for the compare operation.

At step 2810, the size of the data element determines which step is to be executed next. If the size of the data elements is eight bits (packed byte 401 data), then execution unit 130 performs step 2812. However, if the size of the data elements in the packed data is sixteen bits (packed word 402 data), then execution unit 130 performs step 2814. In one embodiment, only eight bit and sixteen bit data element size packed compares are supported. However, in another embodiment, a thirty-two bit data element size packed compare is also supported (packed doubleword 403).

Assuming the size of the data elements is eight bits, then step 2812 is executed. In step 2812, the following is performed. Source1 bits seven through zero are compared to Source2 bits seven through zero generating Result bits seven through zero. Source1 bits fifteen through eight are compared to Source2 bits fifteen through eight generating Result bits fifteen through eight. Source1 bits twenty-three through sixteen are compared to Source2 bits twenty-three through sixteen generating Result bits twenty-three through sixteen. Source1 bits thirty-one through twenty-four are compared to Source2 bits thirty-one through twenty-four generating Result bits thirty-one through twenty-four. Source1 bits thirty-nine through thirty-two are compared to Source2 bits thirty-nine through thirty-two generating Result bits thirty-nine through thirty-two. Source1 bits forty-seven through forty are compared to Source2 bits forty-seven through forty generating Result forty-seven through forty. Source1 bits fifty-five through forty-eight are compared to Source2 bits fifty-five through forty-eight generating Result bits fifty-five through forty-eight. Source1 bits sixty-three through fifty-six are compared to Source2 bits generating Result bits sixty-three through fifty-six.

Assuming the size of the data elements is sixteen bits, then step 2814 is executed. In step 2814, the following is performed. Source1 bits fifteen through zero are compared to Source2 bits fifteen through zero generating Result bits fifteen through zero. Source1 bits thirty-one through sixteen are compared to Source2 bits thirty-one through sixteen generating Result bits thirty-one through

sixteen. Source1 bits forty-seven through thirty-two are compared to Source2 bits forty-seven through thirty-two generating Result bits forty-seven through thirty-two. Source1 bits sixty-three through forty-eight are compared to Source2 bits sixty-three through forty-eight generating Result bits sixty-three through forty-eight.

In one embodiment, the compares of step 2812 are performed simultaneously. However, in another embodiment, these compares are performed serially. In another embodiment, some of these compares are performed simultaneously and some are performed serially. This discussion also applies to the compares of step 2814 as well.

At step 2820, the Result is stored in the DEST 605 register.

Table 37 illustrates the in-register representation of packed compare unsigned greater than operation. The first row of bits is the packed data representation of Source1. The second row of bits is the data representation of Source2. The third row of bits is the packed data representation of the Result. The number below each data element bit is the data element number. For example, Source1 data element three is 10000000_2.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
7	6	5	4	3	2	1	0
>	>	>	>	>	>	>	>
00000000	00000000	10000000	00000000	11110011	00000000	10001110	10001000
↓	↓	↓	↓	↓	↓	↓	↓
11111111	11111111	00000000	11111111	00000000	11111111	11111111	00000000
7	6	5	4	3	2	1	0

Table 37

Table 38 illustrates the in-register representation of packed compare signed greater than or equal to operation on packed byte data.

00101010	01010101	01010101	11111111	10000000	01110000	10001111	10001000
7 >=	6 >=	5 >=	4 >=	3 >=	2 >=	1 >=	0 >=
00000000	00000000	10000000	00000000	11110011	00000000	10001110	10001000
↓	↓	↓	↓	↓	↓	↓	↓
11111111	11111111	11111111	00000000	00000000	11111111	00000000	11111111
7	6	5	4	3	2	1	0

Table 38

PACKED DATA COMPARE CIRCUITS

In one embodiment, the compare operation can occur on multiple data elements in the same number of clock cycles as a single compare operation on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used. That is, registers are simultaneously instructed to perform the compare operation on the data elements. This is discussed in more detail below.

Figure 29 illustrates a circuit for performing packed compare operations on individual bytes of packed data according to one embodiment of the invention. Figure 29 illustrates the use of a modified byte slice compare circuit, byte slice stage$_i$ 2999. Each byte slice, except for the most significant data element byte slice, includes a compare unit and bit control. The most significant data element byte slice need only have a compare unit.

Compare unit$_i$ 2911 and compare unit$_{i+1}$ 2971 each allow eight bits from Source1 to be compared to the corresponding eight bits from Source2. In one embodiment, each compare unit operates like a known eight bit compare circuit. Such a known eight bit compare circuit includes a byte slice circuit allowing the subtraction of Source2 from Source1. The results of the subtraction are processed to determine the results of the compare operation. In one embodiment, the results of the subtraction include an overflow information. This overflow information is tested to determine whether the result of the compare operation is true.

Each compare unit has a Source1 input, a Source2 input, a control input, a next stage signal, a last stage signal, and a result output. Therefore, compare unit$_i$ 2911 has Source1$_i$ 2931 input, Source2$_i$ 2933 input, control$_i$ 2901 input, next

stage$_i$ 2913 signal, last stage$_i$ 2912 input, and a result stored in result register$_i$ 2951. Therefore, compare unit$_{i+1}$ 2971 has Source1$_{i+1}$ 2932 input, Source2$_{i+1}$ 2934 input, control$_{i+1}$ 2902 input, next stage$_{i+1}$ 2973 signal, last stage$_{i+1}$ 2972 input, and a result stored in result register$_{i+1}$ 2952.

The Source1$_n$ input is typically an eight bit portion of Source1. The eight bits represents the smallest type of data element, one packed byte 401 data element. Source2 input is the corresponding eight bit portion of Source2. Operation control 2900 transmits control signals to enable each compare unit to perform the required compare. The control signals are determined from the relationship for the compare (e.g. signed greater than) and the size of the data element (e.g. byte or word). The next stage signal is received from the bit control for that compare unit. Compare units are effectively combined by the bit control units when a larger than byte size data element is used. For example, when the word packed data is compared, the bit control unit between the first compare unit and the second compare unit will cause the two compare units to act as one sixteen bit compare unit. Similarly, the compare unit between the third and fourth compare units will cause these two compare units to act as one compare unit. This continues for the four packed word data elements.

Depending on the desired relationship and the values of Source1 and Source2, the compare unit performs the compare by allowing result of the higher order compare unit to be propagated down to the lower order compare unit or vice versa. That is, each compare unit will provide the results of the compare using the information communicated by the bit control$_i$ 2920. If double word packed data is used, then four compare units act together to form one thirty-two bit long compare unit for each data element. The result output of each compare unit represents the result of the compare operation on the portion of Source1 and Source2 the compare unit is operating upon.

Bit control$_i$ 2920 is enabled from operation control 2900 via packed data enable$_i$ 2906. Bit control$_i$ 2920 controls next stage$_i$ 2913 and last stage$_{i+1}$ 2972. Assume, for example, compare unit$_i$ 2911 is responsible for the eight least significant bits of Source1 and Source2, and compare unit$_{i+1}$ 2971 is responsible for the next eight bits of Source1 and Source2. If a compare on packed byte data is performed, bit control$_i$ 2920 will not allow the result information from compare unit$_{i+1}$ 2971 to be communicated with the compare unit$_i$ 2911, and vice

versa. However, if a compare on packed words is performed, then bit control$_i$ 2920 will allow the result (in one embodiment, an overflow) information from compare unit$_i$ 2911 to be communicated to compare unit$_{i+1}$ and result (in one embodiment, an overflow) information from compare unit$_{i+1}$ 2971 to be communicated to compare unit$_i$ 2911.

For example, in Table 39, a packed byte signed greater than compare is performed. Assume that compare unit$_{i+1}$ 2971 operates on data element one, and compare unit$_i$ 2911 operates on data element zero. Compare unit$_{i+1}$ 2971 compares the most significant eight bits of a word and communicates the result information via last stage$_{i+1}$ 2972. Compare unit$_i$ 2911 compares the least significant eight bits of the word and communicates the result information via next stage$_i$ 2913. However operation control 2900 will cause bit control$_i$ 2920 to stop the propagation of that result information, received from the last stage$_{i+1}$ 2972 and next stage$_i$ 2913, between the compare units.

...	...	...	...	...	...	00001110	00001000
7	6	5	4	3	2	1	0
>	>	>	>	>	>	>	>
...	...	...	...	...	...	00001110	10001000
↓	↓	↓	↓	↓	↓	↓	↓
...	...	...	...	...	...	00000000	11111111
7	6	5	4	3	2	1	0

Table 39

However, if a packed word signed greater than compare is performed, then the result of compare unit$_{i+1}$ 2971 will be communicated to the compare unit$_i$ 2911, and vice versa. Table 40 illustrates this result. This type of communication would be allowed for packed doubleword compares as well.

...	...	...	00001110 00001000
3	2	1	0
>	>	>	>
...	...	...	00001110 10000001
↓	↓	↓	↓
...	...	...	00000000 00000000
3	2	1	0

Table 40

Each compare unit is optionally coupled to a result register. The result register temporarily stores the result of the compare operation until the complete result, Result[63:0] 2960, can be transmitted to the DEST 605 register.

For a complete sixty-four bit packed compare circuit, eight compare units and seven bit control units are used. Such a circuit can also be used to perform a compare on sixty-four bit unpacked data, thereby using the same circuit to perform the unpacked compare operation and the packed compare operation.

ADVANTAGES OF INCLUDING THE DESCRIBED PACKED COMPARE OPERATION IN THE INSTRUCTION SET

The described packed compare instruction stores the result of comparing Source1 and Source2 as a packed mask. As previously described, conditional branches on data are unpredictable, and thus cost processor performance because they break the branch prediction algorithms. However, by generating a packed masked, this comparison instruction reduces the number of required conditional branches based on data. For example, the function (if Y > A then X = X + B; else X = X) may be performed on packed data as shown below in Table 41 (the values shown in Table 41 are shown in hexadecimal notation).

Compare.Greater_Than Source1,Source2

00000001	00000000	Source1=Y_{0-1}
>	>	
00000000	00000001	Source2=A_{0-1}
=		
FFFFFFFF	00000000	Mask

Packed AND Source3,Mask

00000005	0000000A	Source3=B_{0-1}
>	>	
FFFFFFFF	00000000	Mask
=		
00000005	00000000	Result

Packed Add Source4, Result

00000010	00000020	Source4=X_{0-1}
>	>	
00000005	00000000	Result
=		
00000015	00000020	New X_{0-1} value

Table 41

As can be seen from the above example, conditional branches are no longer required. Since a branch instruction is not required, processors that speculatively predict branches do not have a performance decrease when using this compare instruction to perform this and other similar operations. Thus, by providing this compare instruction in the instruction set supported by processor 109, processor 109 can perform algorithms requiring this functionality at a higher performance level.

EXAMPLE MULTIMEDIA ALGORITHMS

To illustrate the versatility of the disclosed instruction set, several example multimedia algorithms are described below. In some cases, similar packed data

instructions could be used to perform certain steps in these algorithms. A number of steps requiring the use of general purpose processor instructions to manage data movement, looping, and conditional branching have been omitted in the following examples.

1) Multiplication of Complex Numbers

The disclosed multiply-add instruction can be used to multiply two complex numbers in a single instruction as shown in Table 42a. The multiplication of two complex number (e.g., r_1 i_1 and r_2 i_2) is performed according to the following equation:

Real Component = $r_1 \cdot r_2 - i_1 \cdot i_2$

Imaginary Component = $r_1 \cdot i_2 + r_2 \cdot i_1$

If this instruction is implemented to be completed every clock cycle, the invention can multiply two complex numbers every clock cycle.

Multiply-Add Source1, Source2

r_1	i_2	r_1	i_1	Source1
r_2	$-i_2$	i_2	r_2	Source2
		=		
Real Component: $r_1r_2-i_1i_2$		Imaginary Component: $r_1i_2+r_2i_1$		Result 1

Table 42a

As another example, Table 42b shows the instructions used to multiply together three complex numbers.

Multiply-Add Source1, Source2

r_1	i_1	r_1	i_1	Source1
r_2	$-i_2$	i_2	r_2	Source2
		=		
Real Component1: $r_1r_2-i_1i_2$		Imaginary Component1: $r_1i_2+r_2i_1$		Result1

Packed Shift Right Source1, Source2

Real Component1			Imaginary Component1	Result1
16				
=				
	Real Component1		Imaginary Component1	Result2

Pack Result2, Result2

	Real Component1		Imaginary Component1	Result2
	Real Component1		Imaginary Component1	Result2
=				
Real Component1	Imaginary Component1	Real Component1	Imaginary Component1	Result3

Multiply-Add Result3, Source3

Real Component1: r1r2-i1i2	Imaginary Component1: r1i2+r2i1	Real Component1: r1r2-i1i2	Imaginary Component1: r1i2+r2i1	Result3
r3	-i3	i3	r3	Source3
=				
Real Component2		Imaginary Component2		Result4

Table 42b

2) Multiply Accumulation Operations

The disclosed instructions can also be used to multiply and accumulate values. For example, two sets of four data elements (A_{1-4} and B_{1-4}) may be multiplied and accumulated as shown below in Table 43. In one embodiment,

each of the instructions shown in Table 43 is implemented to complete each clock cycle.

Multiply-Add Source1, Source2

0	0	A_1	A_2	Source1
0	0	B_1	B_2	Source2
		=		
0		$A_1B_1+A_2B_2$		Result1

Multiply-Add Source3, Source4

0	0	A_3	A_4	Source3
0	0	B_3	B_4	Source4
		=		
0		$A_3A_4+B_3B_4$		Result2

Unpacked Add Result1, Result2

0	$A_1B_1+A_2B_2$	Result1
0	$A_3A_4+B_3B_4$	Result2
	=	
0	$A_1B_1+A_2B_2+A_3A_4+B_3B_4$	Result3

Table 43

If the number of data elements in each set exceeds 8 and is a multiple of 4, the multiplication and accumulation of these sets requires fewer instructions if performed as shown in Table 44 below.

Multiply-Add Source1, Source2

A_1	A_2	A_3	A_4	Source1
B_1	B_2	B_3	B_4	Source2
=				
$A_1B_1+A_2B_2$		$A_3B_3+A_4B_4$		Result1

Multiply-Add Source3, Source4

A_5	A_6	A_7	A_8	Source3
B_5	B_6	B_7	B_8	Source4
=				
$A_5B_5+A_6B_6$		$A_7B_7+A_8B_8$		Result2

Packed Add Result1, Result2

$A_1B_1+A_2B_2$	$A_3B_3+A_4B_4$	Result1
$A_5B_5+A_6B_6$	$A_7B_7+A_8B_8$	Result2
=		
$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3

Unpack High Result3, Source5

$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3
0	0	Source5
=		
0	$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	Result4

Unpack Low Result3, Source5

$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3
0	0	Source5
=		
0	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result5

Packed Add Result4, Result5

0	$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	Result4
0	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result5
=		
0	TOTAL	Result6

Table 44

As another example, Table 45 shows the separate multiplication and accumulation of sets A and B and sets C and D, where each of these sets includes 2 data elements.

Multiply-Add Source1, Source2

A_1	A_2	C_1	C_2	Source1
B_1	B_2	D_1	D_2	Source2
=				
$A_1B_1+A_2B_2$		$C_1D_1+C_2D_2$		Result1

Table 45

As another example, Table 46 shows the separate multiplication and accumulation of sets A and B and sets C and D, where each of these sets includes 4 data elements.

Multiply-Add Source1, Source2

A_1	A_2	C_1	C_2	Source1
B_1	B_2	D_1	D_2	Source2
		=		
$A_1B_1+A_2B_2$		$C_1D_1+C_2D_2$		Result1

Multiply-Add Source3, Source4

A_3	A_4	C_3	C_4	Source3
B_3	B_4	D_3	D_4	Source4
		=		
$A_3B_3+A_4B_4$		$C_3D_3+C_4D_4$		Result2

Packed Add Result1, Result2

$A_1B_1+A_2B_2$	$C_1D_1+C_2D_2$	Result1
$A_3B_3+A_4B_4$	$C_3D_3+C_4D_4$	Result2
	=	
$A_1B_1+A_2B_2+A_3B_3+A_4B_4$	$C_1D_1+C_2D_2+C_3D_3+C_4D_4$	Result6

Table 46

3) Dot Product Algorithms

Dot product (also termed as inner product) is used in signal processing and matrix operations. For example, dot product is used when computing the product of matrices, digital filtering operations (such as FIR and IIR filtering), and computing correlation sequences. Since many speech compression algorithms (e.g., GSM, G.728, CELP, and VSELP) and Hi-Fi compression algorithms (e.g., MPEG and subband coding) make extensive use of digital filtering and

correlation computations, increasing the performance of dot product increases the performance of these algorithms.

The dot product of two length N sequences A and B is defined as:

$$\text{Result} = \sum_{i=0}^{N-1} Ai \cdot Bi$$

Performing a dot product calculation makes extensive use of the multiply accumulate operation where corresponding elements of each of the sequences are multiplied together, and the results are accumulated to form the dot product result.

By including the move, packed add, multiply-add, and pack shift operations, the invention allows the dot product calculation to be performed using packed data. For example if the packed data type containing four sixteen-bit elements is used, the dot product calculation may be performed on two sequences each containing four values by:

1) accessing the four sixteen-bit values from the *A* sequence to generate Source1 using a move instruction;
2) accessing four sixteen-bit values from the *B* sequence to generate Source2 using a move instruction; and
3) multiplying and accumulating as previously described using a multiply-add, packed add, and shift instructions.

For vectors with more than just a few elements the method shown in Table 46 is used and the final results are added together at the end. Other supporting instructions include the packed OR and XOR instructions for initializing the accumulator register, the packed shift instruction for shifting off unwanted values at the final stage of computation. Loop control operations are accomplished using instructions already existing in the instruction set of processor 109.

4) 2-Dimensional Loop Filter

2-dimensional loop filters are used in certain multimedia algorithms. For example, the filter coefficients shown below in Table 47 may be used in video conferencing algorithms to perform a low pass filter on pixel data.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Table 47

To calculate the new value of a pixel at location (x, y), the following equation is used:

$$\text{Resulting Pixel} = (x-1,y-1) + 2(x,y-1) + (x+1,y-1) + 2(x-1,y) + 4(x,y) + 2(x+1,y) + (x-1,y+1) + 2(x,y+1) + (x+1,y+1)$$

By including the pack, unpack, move, packed shift, and a packed add, the invention allows a 2-dimensional loop filter to be performed using packed data. According to one implementation of the previously described loop filter, this loop filter is applied as two simple 1-dimensional filters -- i.e., the above 2-dimensional filter can be applied as two 121 filters. The first filter is in the horizontal direction, while the second filter is in the vertical direction.

Table 48 shows a representation of an 8x8 block of pixel data.

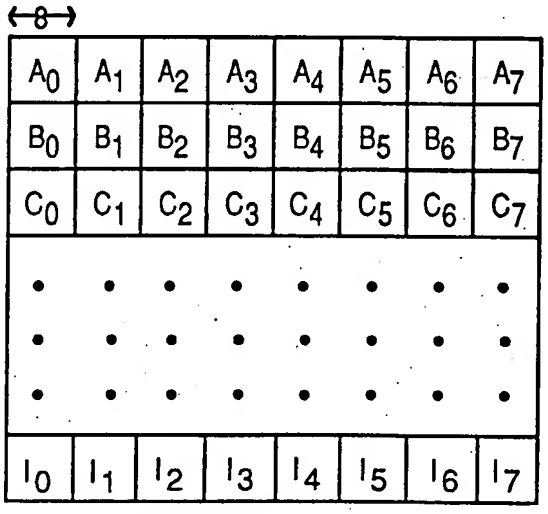


Table 48

The following steps are performed to implement the horizontal pass of the filter on this 8x8 block of pixel data:

1) accessing eight 8-bit pixel values as packed data using a move instruction;
2) unpacking the eight 8-bit pixels into a 16-bit packed data containing four 8-bit pixels (Source1) to maintain accuracy during accumulations;
3) duplicating Source1 two times to generate Source2 and Source3;
4) performing an unpacked shift right by 16 bits on Source1;
5) performing an unpacked shift left by 16 bits on Source 3;
6) generating (Source1 + 2*Source2 + Source3) by performing the following packed adds:
   a) Source1 = Source1 + Source2,
   b) Source1 = Source1 + Source2,
   c) Source1 = Source1 + Source3;
7) storing the resulting packed word data as part of an 8x8 intermediate result array; and
8) repeating these steps until the entire 8x8 intermediate result array is generated as shown in Table 49 below (e.g., IA_0 represents the intermediate result for A_0 from Table 49).

←16→

IA_0	IA_1	IA_2	IA_3	IA_4	IA_5	IA_6	IA_7
IB_0	IB_1	IB_2	IB_3	IB_4	IB_5	IB_6	IB_7
IC_0	IC_1	IC_2	IC_3	IC_4	IC_5	IC_6	IC_7
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•
II_0	II_1	II_2	II_3	II_4	II_5	II_6	II_7

Table 49

The following steps are performed to implement the vertical pass of the filter on the 8x8 intermediate result array:

1) accessing a 4x4 block of data from the intermediate result array as packed data using a move instruction to generate Source1, Source2, and Source3 (e.g., see Table 50 for an example);

←16→

IA_0	IA_1	IA_2	IA_3	Source1
IB_0	IB_1	IB_2	IB_3	Source2
IC_0	IC_1	IC_2	IC_3	Source3

Table 50

2) generating (Source1 + 2*Source2 + Source 3) by performing the following packed adds:

a) Source1 = Source1 + Source2,

b) Source1 = Source1 + Source2,

c) Source1 = Source1 + Source3;

3) performing a packed shift right by 4 bits on the resulting Source1 to generate the sum of the weights -- this is effectively dividing by 16;

4) packing the resulting Source 1 with saturation to convert the 16-bit values back into 8-bit pixel values;

6) storing the resulting packed byte data as part of an 8x8 result array (in regards to the example shown in Table 50, these four bytes represent the new pixel values for B_0, B_1, B_2, and B_3); and

7) repeating these steps until the entire 8x8 result array is generated.

It is worth while to note, that the top and bottom rows of the 8x8 result array are determined using a different algorithm that is not described here so not to obscure the invention.

Thus, by providing on processor 109 the pack, unpack, move, packed shift, and packed add instructions, the invention provides for a significant performance increase over prior art general processors which must perform the operations required by such filters 1 data element at a time.

5) Motion Estimation

Motion estimation is used in several multimedia applications (e.g., video conferencing and MPEG (high quality video playback)). In regard to video conferencing, motion estimation is used to reduce the amount of data which must

be transmitted between terminals. Motion estimation works by dividing the video frames into fixed size video blocks. For each block in Frame1, it is determined whether there is a block containing a similar image in Frame2. If such a block is contained in Frame2, that block can be described with a motion vector reference into Frame1. Thus, rather than transmitting all of the data representing that block, only a motion vector need be transmitted to the receiving terminal. For example, if a block in Frame1 is similar to and at the same screen coordinates as a block in Frame2, only a motion vector of 0 need to sent for that block. However, if a block in Frame1 is similar to, but at different screen coordinates than, a block in Frame2, only a motion vector indicating the new location of that block need be sent. According to one implementation, to determine if a block A in Frame1 is similar to a block B in Frame2, the sum of the absolute differences between the pixel values is determined. The lower the summation, the more similar block A is to block B (i.e., if the summation is 0, block A is identical to block B).

By including the move, unpack, packed add, packed subtract with saturate, and logical operations, the invention allows motion estimation to be performed using packed data. For example, if two 16x16 blocks of video are represented by two arrays of 8-bit pixel values stored as packed data, the absolute difference of the pixel values in these blocks may be calculated by:

1) accessing eight 8-bit values from block A to generate Source1 using a move instruction;
2) accessing eight 8-bit values from block B to generate Source2 using a move instruction;
3) performing a packed subtract with saturate to subtract Source1 from Source2 generating Source 3 -- By subtracting with saturate, Source 3 will contain only the positive results of this subtraction (i.e., the negative results will be zeroed);
4) performing a packed subtract with saturate to subtract Source2 from Source1 generating Source 4 -- By subtracting with saturate, Source 4 will contain only the positive results of this subtraction (i.e., the negative results will be zeroed);

5) performing a packed OR operation on Source3 and Source4 to produce Source5 -- By performing this OR operation, Source5 contains the absolute value of Source1 and Source2;

6) repeating these steps until the 16x16 blocks have been processed.

The resulting 8-bit absolute values are unpacked into 16-bit data elements to allow for 16-bit precision, and then summed using packed adds.

Thus, by providing on processor 109 move, unpack, packed add, packed subtract with saturate, and logical operations, the invention provides for a significant performance increase over prior art general purpose processors which must perform the additions and the absolute differences of the motion estimation calculation one data element at a time.

6) Discrete Cosign Transform

Discrete Cosine Transform (DCT) is a well known function used in many signal processing algorithms. Video and image compression algorithms, in particular, make extensive use of this transform.

In image and video compression algorithms, DCT is used to transform a block of pixels from the spatial representation to the frequency representation. In the frequency representation, the picture information is divided into frequency components, some of which are more important than others. The compression algorithm selectively quantizes or discards the frequency components that do not adversely affect the reconstructed picture contents. In this manner, compression is achieved.

There are many implementations of the DCT, the most popular being some kind of fast transform method modeled based on the Fast Fourier Transform (FFT) computation flow. In the fast transform, an order N transform is broken down to a combination of order N/2 transforms and the result recombined. This decomposition can be carried out until the smallest order 2 transform is reached. This elementary 2 transform kernel is often referred to as the butterfly operation. The butterfly operation is expressed as follows:

$$X = a*x + b*y$$

$$Y = c*x - d*y$$

where a, b, c and d are termed the coefficients, x and y are the input data, and X and Y are the transform output.

By including the move, multiply-add, and packed shift operations, the invention allows the DCT calculation to be performed using packed data in the following manner:

1) accessing the two 16-bit values representing x and y to generate Source1 (see Table 51 below) using the move and unpack instructions;

2) generating Source2 as shown in Table 51 below -- Note that Source2 may be reused over a number of butterfly operations; and

3) performing a multiply-add instruction using Source1 and Source2 to generate the Result (see Table 51 below).

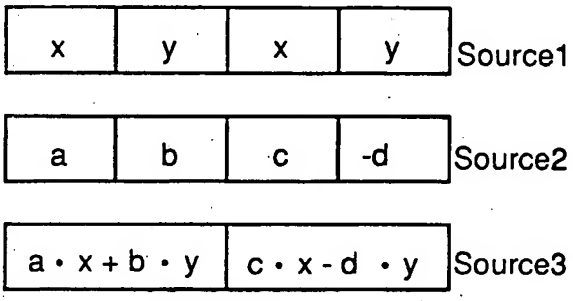


Table 51

In some situations, the coefficients of the butterfly operation are 1. For these cases, the butterfly operation degenerates into just adds and subtracts that may be performed using the packed add and packed subtract instructions.

An IEEE document specifies the accuracy with which inverse DCT should be performed for video conferencing. (See, IEEE Circuits and Systems Society, "IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform," IEEE Std. 1180-1990, IEEE Inc. 345 East 47th St., NY, NY 10017, USA, March 18, 1991). The required accuracy is met by the disclosed multiply-add instruction because it uses 16-bit inputs to generate 32-bit outputs.

Thus, by providing on processor 109 the move, multiply-add, and packed shift operations, the invention provides for a significant performance increase over prior art general purpose processors which must perform the additions and the multiplications of the DCT calculation one data element at a time.

ALTERNATIVE EMBODIMENTS

While the invention has been described in which each of the different operations have separate circuitry, alternative embodiments could be implemented such that certain circuitry is shared by different operations. For example, in one embodiment the following circuitry is used: 1) a single arithmetic logic unit (ALU) to perform the packed add, packed subtract, packed compare, and packed logical operations; 2) a circuitry unit to perform the pack, unpack, and packed shift operations; 3) a circuitry unit to perform the packed multiply and multiply-add operations; and 4) a circuitry unit to perform the population count operation.

The terms corresponding and respective are used herein to refer to the predetermined relationship between the data elements stored in two or more packed data. In one embodiment, this relationship is based on the bit positions of the data elements in the packed data. For example, data element 0 (e.g., stored in bit positions 0-7 in packed byte format) of a first packed data corresponds to data elements 0 (e.g., stored in bit positions 0-7 in packed byte format) of a second packed data. However, this relationship may differ in alternative embodiments. For example, corresponding data elements in the first and second packed data may be of different sizes. As another example, rather than the least significant data element of a first packed data corresponding to the least significant data element of a second packed data (and so on), the data elements in the first and second packed data may correspond to each other in some other order. As another example, rather than having a 1 to 1 correspondence of data elements in the first and second packed data, the data elements may correspond at a different ratio (e.g., the first packed data may have one or more data elements which correspond to two or more different data elements in a second packed data).

While the invention has been described in terms of several embodiments, those skilled in the art will recognize that the invention is not limited to the embodiments described. The method and apparatus of the invention can be practiced with modification and alteration within the spirit and scope of the appended claims. The description is thus to be regarded as illustrative instead of limiting on the invention.

THE CLAIMS

What is claimed is:

1. A computer system comprising:

a processor including a first register; and

a storage area coupled to said processor having stored therein,

a pack instruction operating on a first packed data and a second packed data, said first packed data containing at least a first data element and a second data element, said second packed data containing at least a third data element and a fourth data element, each of said first data element, said second data element said third data element, and said fourth data element containing a set of bits, said processor packing a portion of each of said first data element, said second data element, said third data element, and said fourth data element to form a third packed data in response to receiving said pack instruction ;

an unpack instruction operating on a fourth packed data and a fifth packed data, said fourth packed data containing at least a fifth data element and a sixth data element, said fifth packed data containing at least a seventh data element corresponding to said fifth data element and a eighth data element corresponding to said sixth data element, each of said fifth data element, said sixth data element, said seventh data element and said eighth data element including a set of bits, said processor generating a sixth packed data containing at least said fifth data element from said fourth packed data and said seventh data element from said fifth packed data in response to receiving said unpack instruction;

a packed add instruction, said processor separately adding together in parallel corresponding data elements of said fourth packed data and said fifth packed data in response to receiving said packed add instruction;

a packed subtract instruction, said processor separately subtracting in parallel corresponding data elements of said fourth packed data and said fifth packed data in response to receiving said packed subtract instruction;

a packed shift instruction, said processor separately shifting in parallel at least said first data element by an indicated count and said second data

element by an indicated count in response to receiving said packed shift instruction; and

a packed compare instruction, said processor separately comparing in parallel corresponding data elements from said fourth packed data and said fifth packed data according to an indicated relationship and storing as a result a packed mask in said first register in response to receiving said packed compare instruction, said packed mask containing at least a first mask element and a second mask element each including said predetermined number of bits, each bit in said first mask element indicating said result of comparing said fifth data element in said fourth packed data to said seventh data element in said fifth packed data, each bit in said second mask element indicating said result of comparing said sixth data element in said fourth packed data to said eighth data element in said fifth packed data.

2. The computer system of claim 1, said storage area further having stored therein a packed multiply instruction, said processor separately multiplying together in parallel corresponding data elements of said fourth packed data and said fifth packed data in response to receiving said packed multiply instruction.

3. The computer system of claim 1, said storage area further having stored therein a packed multiply-add instruction, said processor multiplying together said first data element and said third data element to generate a first intermediate result, multiplying together said second data element and said fourth data element to generate a second intermediate result, and adding together said first intermediate result and said second intermediate result to generate a ninth data element in a final result in response to receiving said packed multiply-add instruction.

4. The computer system of claim 2, wherein said first data element, said second data element, said third data element, and said fourth data element each include a predetermined number of bits, and wherein said ninth data element includes two times said predetermined number of bits.

5. The computer system of claim 1, said storage area further having stored therein a population count instruction, said processor determining in parallel how many bits in said first data element are set to a predetermined value and how many bits in said second data element are set to said predetermined value in response to receiving said population count instruction.

6. The computer system of claim 1, said storage area further having stored therein,

a first packed logical instruction, said processor logically ANDing together in parallel corresponding data elements from said fourth packed data and said fifth packed data in response to receiving said first packed logical instruction;

a second packed logical instruction, said processor logically ANDing in parallel data elements from said fourth packed data with the logical inversion of corresponding data elements from said fifth packed data in response to receiving said second packed logical instruction;

a third packed logical instruction, said processor logically ORing together in parallel corresponding data elements from said fourth packed data and said fifth packed data in response to receiving said third packed logical instruction; and

a fourth packed logical instruction, said processor logically exclusive ORing together in parallel corresponding data elements from said fourth packed data and said fifth packed data in response to receiving said fourth packed logical instruction.

7. The computer system of Claim 1, wherein said shifting is arithmetic.

8. The computer system of Claim 1, wherein said shifting is logical.

9. The computer system of Claim 1, wherein said shifting is in a rightward direction.

10. The computer system of Claim 1, wherein said shifting is in a leftward direction.

11. The computer system of Claim 1, wherein said first data element, said second data element, said third data element, and said fourth data element each include a predetermined number of bits.

12. The computer system of Claim 11, wherein said portion includes half of said predetermined number of bits.

13. The computer system of Claim 1, wherein said fifth data element, said sixth data element, said seventh data element, and said eighth data element each include said predetermined number of bits.

14. A method for manipulating a first packed data and a second packed data, said first packed data containing at least a first data element and a second data element, said second packed data containing at least a third data element corresponding to said first data element and a fourth data element corresponding to said second data element, each of said first data element, said second data element, said third data element and said fourth data element each containing a set of bits, said method comprising the computer implemented steps of:

receiving an instruction;

determining if said instruction is one of a pack instruction, an unpack instruction, a packed add instruction, a packed subtract instruction, a packed shift instruction, and a packed compare instruction;

if said instruction is said pack instruction, then packing a portion of each of said first data element, said second data element, said third data element, and said fourth data element to form a third packed data;

if said instruction is said unpack instruction, generating a fourth packed data containing at least said first data element from said first packed data and said third data element from said second packed data;

if said instruction is said packed add instruction, separately adding together in parallel corresponding data elements of said first packed data and said second packed data;

if said instruction is said packed subtract instruction, separately subtracting in parallel corresponding data elements of said first packed data and said second packed data;

if said instruction is said packed shift instruction, separately shifting in parallel at least said first data element and said second data element by an indicated count; and

if said instruction is said packed compare instruction, separately comparing in parallel corresponding data elements from said first packed data and said second packed data according to an indicated relationship and generating as a result a packed mask, said packed mask containing at least a first mask element and a second mask element each including said predetermined number of bits, each bit in said first mask element indicating said result of comparing said first data element in said first packed data to said third data element in said second packed data, each bit in said second mask element indicating said result of comparing said second data element in said first packed data to said fourth data element in said second packed data.

15. The method of claim 14, wherein:

said step of determining further includes determining if said instruction is a packed multiply instruction; and

said method further including the step of:

if said instruction is said packed multiply instruction, separately multiplying together in parallel corresponding data elements of said first packed data and said second packed data.

16. The method of claim 14, wherein:

said step of determining further includes determining if said instruction is a packed multiply-add instruction; and

said method further including the steps of:

if said instruction is said packed multiply-add instruction, performing the steps of:

multiplying together said first data element and said third data element to generate a first intermediate result,

multiplying together said second data element and said fourth data element to generate a second intermediate result, and

adding together said first intermediate result and said second intermediate result to generate a fifth data element in a final result.

17. The method of claim 16, wherein said first data element, said second data element, said third data element, and said fourth data element each contain a predetermined number of bits, and wherein each of said fourth data element and said fifth data element each contain two times said predetermined number of bits.

18. The method of claim 14, wherein:

said step of determining further includes determining if said instruction is a population count instruction; and

said method further including the step of:

if said instruction is said population count instruction, determining in parallel how many bits in said first data element are set to a predetermined value and how many bits in said second data element are set to said predetermined value.

19. The method of claim 14, wherein:

said step of determining further includes determining if said instruction is one of a plurality of packed logical instructions; and

said method further including the step of:

if said instruction is a first of said plurality of packed logical instructions, then logically ANDing together in parallel corresponding data elements from said first packed data and said second packed data;

if said instruction is a second of said plurality of packed logical instructions, logically ANDing together in parallel data elements from said first packed data with the logical inversion of corresponding data elements from said second packed data;

if said instruction is a third of said plurality of packed logical instructions, logically ORing together in parallel corresponding data elements from said first packed data and said second packed data; and

if said instruction is a fourth of said plurality of packed logical instructions, logically exclusive ORing together in parallel corresponding data elements from said first packed data and said second packed data.

20. The computer system of Claim 14, wherein said step of separately shifting is performed as one of an arithmetic shift and a logical shift.

21. The computer system of Claim 14, wherein said step of separately shifting is performed such that both said first data element and said second data element are shifted in one of a rightward direction and a leftword direction.

22. The method of claim 16, wherein said first data element, said second data element, said third data element, and said fourth data element each contain a predetermined number of bits.

23. The method of claim 22, wherein said portion contains half of said predetermined number of bits.

24. The method of claim 14, wherein said step of determining is performed by a decoder.

AMENDED CLAIMS

[received by the International Bureau on 01 November 1996(01.11.96); original claims 1-24 cancelled, new claims 25-55 added, other claims unchanged (8 pages)]

25. A processor comprising:

a storage area configured to contain a first packed data and a second packed data respectively including a first plurality of data elements and a second plurality of data elements, wherein each data element in the first plurality of data elements corresponds to a data element in the second plurality of data elements;

a decoder configured to decode an instruction;

a first circuit, coupled to the storage area and the decoder, the first circuit configured to simultaneously copy, in response to an unpack instruction, certain corresponding data elements of the first and second plurality of data elements into the storage area as a plurality of result data elements in a third packed data;

a second circuit, coupled to the storage area and the decoder, the second circuit configured to simultaneously copy, in response to a pack instruction, a part of each data element in the first and second plurality of data elements into the storage area as a plurality of result data elements in a third packed data;

a third circuit, coupled to the storage area and the decoder, the third circuit configured to simultaneously multiply, in response to a multiply instruction, each data element of the first plurality of data elements with a corresponding data element of the second plurality of data elements to generate a plurality of result data elements in a third packed data, wherein each result data element includes only high order bits or low order bits;

a fourth circuit, coupled to the storage area and the decoder, the fourth circuit configured to simultaneously add, in response to an add instruction, each data element of the first plurality of data elements with a corresponding data element of the second plurality of data elements to generate a plurality of result data elements in a third packed data;

a fifth circuit, coupled to the storage area and the decoder, the fifth circuit configured to simultaneously subtract, in response to a subtract instruction, each data element of the first plurality of data elements from a corresponding data element of the second plurality of data elements to generate a plurality of result data elements in a third packed data;

a sixth circuit, coupled to the storage area and the decoder, the sixth circuit configured to simultaneously compare each data element in the first packed data against a corresponding data element in the second packed data, and the sixth circuit further

configured to generate a packed mask having a plurality of mask elements, each mask element representing a corresponding comparison, and each mask element includes a plurality of bits all having either a first predetermined value or a second predetermined value based on whether the corresponding comparison was TRUE or FALSE; and

a seventh circuit, coupled to the storage area and the decoder, the seventh circuit configured to independently shift, in response to a shift instruction, each data element of the first plurality of data elements by a shift count.

26. The processor of Claim 25 further comprising:

an eighth circuit, coupled to the storage area and the decoder, the eighth circuit configured to simultaneously multiply, in response to a multiply-add instruction, each data element of the first plurality of data elements with a corresponding data element of the second plurality of data elements to generate a plurality of intermediate result data elements, and simultaneously adding adjacent intermediate result data elements to generate a plurality of result data elements in a third packed data.

27. The processor of Claim 25, further comprising a ninth circuit, coupled to the storage area and the decoder, the ninth circuit configured to simultaneously generate, in response to a population count instruction, a result packed data having at least a first and second result data element, the first result data element representing a total number bits set in a first data element of the first plurality of data elements, and the second result data element representing a total number bits set in a second data element of the first plurality of data elements.

28. The processor of Claim 25, further comprising:

a tenth circuit, coupled to the storage area and the decoder, the tenth circuit configured to simultaneously logically AND, in response to a logical AND instruction, each data element of the first plurality of data element with a corresponding data element from the second plurality of data elements to generate a plurality of result data elements in a third packed data;

an eleventh circuit, coupled to the storage area and the decoder, the eleventh circuit configured to simultaneously logically AND, in response to a second logical AND

instruction, an inversion of the each element of the first plurality of data elements with a corresponding data element from the second plurality of data elements to generate a plurality of result data elements in a third packed data;

a twelfth circuit, coupled to the storage area and the decoder, the twelfth circuit configured to simultaneously logically OR, in response to a logical OR instruction, each data element of the first plurality of data elements with a corresponding data element of the second plurality of data elements to generate a plurality of result data elements in a third packed data; and

a thirteenth circuit, coupled to the storage area and the decoder, the thirteenth circuit coupled to simultaneously logically OR, in response to a second logical OR instruction, an inversion of each data element of the first plurality of data elements with a corresponding data element of the second plurality of data elements to generate a plurality of result data elements in a third packed data.

29. The processor of Claim 25, wherein the seventh circuit is further configured fill, in each data element, a shift count number of bits with zeros.

30. The processor of Claim 25, wherein the seventh circuit is further configured to fill, in each data element, a shift count number of bits with a sign bit for the respective data element.

31. The processor of Claim 25, wherein the first circuit is further configured to simultaneously copy half of the data elements in the first plurality of data elements and half of the data elements of the second plurality of data elements.

32. The processor of claim 31, wherein the corresponding data elements copied from the first and second plurality of data elements are copied into the storage area adjacent to each other as the plurality of result data elements.

33) The processor of Claim 32, wherein the first plurality of data elements are copied in the same order as the first plurality of data elements appear in the first packed data.

34) The processor of Claim 25, wherein the part of each data element copied by the second circuit is either the low order bits or the high order bits of each data element in the first and second plurality of data elements.

35) The processor of claim 34, wherein the parts of the first plurality of data elements are copied into the third packed data adjacent to each other as the plurality of result data elements.

36) The processor of claim 35, wherein the parts of the first and second plurality of data elements are copied into the third packed data in the same order as the first and second plurality of data elements appear in the first and second packed data.

37) The processor of Claim 25, wherein the first and second plurality of data elements each include two data elements, each data element representing thirty-two bits.

38) The processor of Claim 25, wherein the first and second plurality of data elements each include four data elements, each data element representing sixteen bits.

39) The processor of Claim 25, wherein the first and second plurality of data elements each include eight data elements, each data element representing eight bits.

40) A processor comprising:

a first storage area configured to contain a first packed data having a first plurality of data elements;

a second storage area configured to contain a second packed data having a second plurality of data elements, each data element of the second plurality of data elements corresponding to a different data element of the first plurality of data elements;

a decoder configured to decode an instruction, the instruction indicating a first address corresponding to the first storage area, a second address corresponding to the

second storage area, a destination address corresponding to a third storage area, and an operation to be performed on at least the first plurality of data elements;

a first circuit, coupled to the storage areas and the decoder, the first circuit configured to copy in parallel, in response to an unpack instruction, certain corresponding data elements from the first and second plurality of data elements into the third storage area as a third plurality of data elements in a third packed data;

a second circuit, coupled to the storage areas and the decoder, the second circuit configured to copy in parallel, in response to a pack instruction, a part of each data element in the first and second plurality of data elements into the third storage area as a third plurality of data elements in a third packed data;

a third circuit, coupled to the storage areas and the decoder, the third circuit configured to multiply in parallel, in response to a multiply instruction, each data element of the first plurality of data elements with a different corresponding data element of the second plurality of data elements to generate a third plurality of data elements in a third packed data, wherein each data element of the third plurality of data elements includes only high order bits or low order bits;

a fourth circuit, coupled to the storage areas and the decoder, the fourth circuit configured to add in parallel, in response to an add instruction, each data element of the first plurality of data elements with a different corresponding data elements of the second plurality of data elements to generate a third plurality of data elements in a third packed data;

a fifth circuit, coupled to the storage areas and the decoder, the fifth circuit configured to subtract in parallel, in response to a subtract instruction, each data element of the first plurality of data element from a different corresponding data element of the second plurality of data elements to generate a third plurality of data elements in a third packed data;

a sixth circuit, coupled to the storage area and the decoder, the sixth circuit configured to compare in parallel, in response to a compare instruction, each data element in the first packed data against a different corresponding data element in the second packed data, and the sixth circuit further configured to generate a packed mask having a plurality of mask elements, each mask element in the plurality of mask elements representing a corresponding comparison generated by the sixth circuit comparing in parallel each data element in the first packed data against a different corresponding data element in the second packed data, each mask element in the plurality of mask elements including a

plurality of bits all having either a first predetermined value or a second predetermined value based on whether the corresponding comparison was TRUE or FALSE; and

an seventh circuit, coupled to the storage areas and the decoder, the seventh circuit configured to independently shift, in response to a shift instruction, each data element of the first plurality of data elements by a shift count.

41) The processor of Claim 40 further comprising:

an eighth circuit, coupled to the storage areas and the decoder, the eighth circuit configured to multiply in parallel, in response to a multiply-add instruction, each data element of the first plurality of data elements with a different corresponding data element of the second plurality of data elements to generate a third plurality of data elements, and adding, in parallel, adjacent data elements in the third plurality of data elements to generate a fourth plurality of data elements in a third packed data.

42) The processor of Claim 41, further comprising a ninth circuit, coupled to the storage areas and the decoder, the ninth circuit configured to generate in parallel, in response to a population count instruction, a result packed data having at least a first and second result data element, the first result data element representing a total number bits set in a first data element of the first plurality of data elements, and the second result data element representing a total number bits set in a second data element of the first plurality of data elements.

43) The processor of Claim 42, further comprising:

a tenth circuit, coupled to the storage areas and the decoder, the tenth circuit configured to logically AND in parallel, in response to a logical AND instruction, each data element of the first plurality of data elements with a different corresponding data element from the second plurality of data elements to generate a third plurality of data elements in a third packed data;

an eleventh circuit, coupled to the storage areas and the decoder, the eleventh circuit configured to logically AND in parallel, in response to a second logical AND instruction, an inversion of each data element of the first plurality of data elements with a

different corresponding data element from the second plurality of data elements to generate a third plurality of data elements in a third packed data;

a twelfth circuit, coupled to the storage areas and the decoder, the twelfth circuit configured to logically OR in parallel, in response to a logical OR instruction, each data element of the first plurality of data elements with a different corresponding data element of the second plurality of data elements to generate a third plurality of data elements in a third packed data; and

a thirteenth circuit, coupled to the storage areas and the decoder, the thirteenth circuit coupled to logically OR in parallel, in response to a second logical OR instruction, an inversion of each data element of the first plurality of data elements with a different corresponding data element of the second plurality of data elements to generate a third plurality of data elements in a third packed data.

44) The processor of Claim 40, wherein the seventh circuit is further configured to logically shift in parallel, in response to a logical shift instruction, each data element of the first plurality of data elements, wherein a shift count number of bits in each data element is filled with zeros.

45) The processor of Claim 40, wherein the seventh circuit is further configured to independently perform an arithmetic shift in parallel, in response to a arithmetic shift instruction, on each data element of the first plurality of data elements, wherein a shift count number of bits in each data element is filled with a sign bit for the respective data element.

46) The processor of Claim 40, wherein the first circuit is further configured to copy in parallel, in response to the unpack instruction, half of the data elements in the first plurality of data elements and half the data elements of the second plurality of data elements.

47) The processor of claim 46, wherein corresponding data elements copied from the first and second plurality of data elements, by the first circuit are placed adjacent to each other, into the third storage area as the third plurality of data elements.

48) The processor of Claim 47, wherein each of the first plurality of data elements copied, by the first circuit, are copied into the third packed data in the same order as the first plurality of data elements appear in the first packed data.

49) The processor of Claim 40, wherein the part of each data element copied, by the second circuit, is half of the bits in each data element in the first and second plurality of data elements.

50) The processor of Claim 49, wherein the part of each data element copied, by the second circuit, is either the low or the high order bits of each data element in the first and second plurality of data elements.

51) The processor of Claim 50, wherein the parts copied from data elements in the first plurality of data elements, by the second circuit, are placed adjacent in the third plurality of data elements.

52) The processor of claim 51, wherein the parts copied from the first and second plurality of data elements are copied into the third packed data in the same order as the first and second plurality of data elements appear in the first and second packed data.

53) The processor of Claim 40, wherein the first and second plurality of data elements each include two data elements, each data element representing thirty-two bits.

54) The processor of Claim 40, wherein the first and second plurality of data elements each include four data elements, each data element representing sixteen bits.

55) The processor of Claim 40, wherein the first and second plurality of data elements each include eight data elements, each data element representing eight bits.

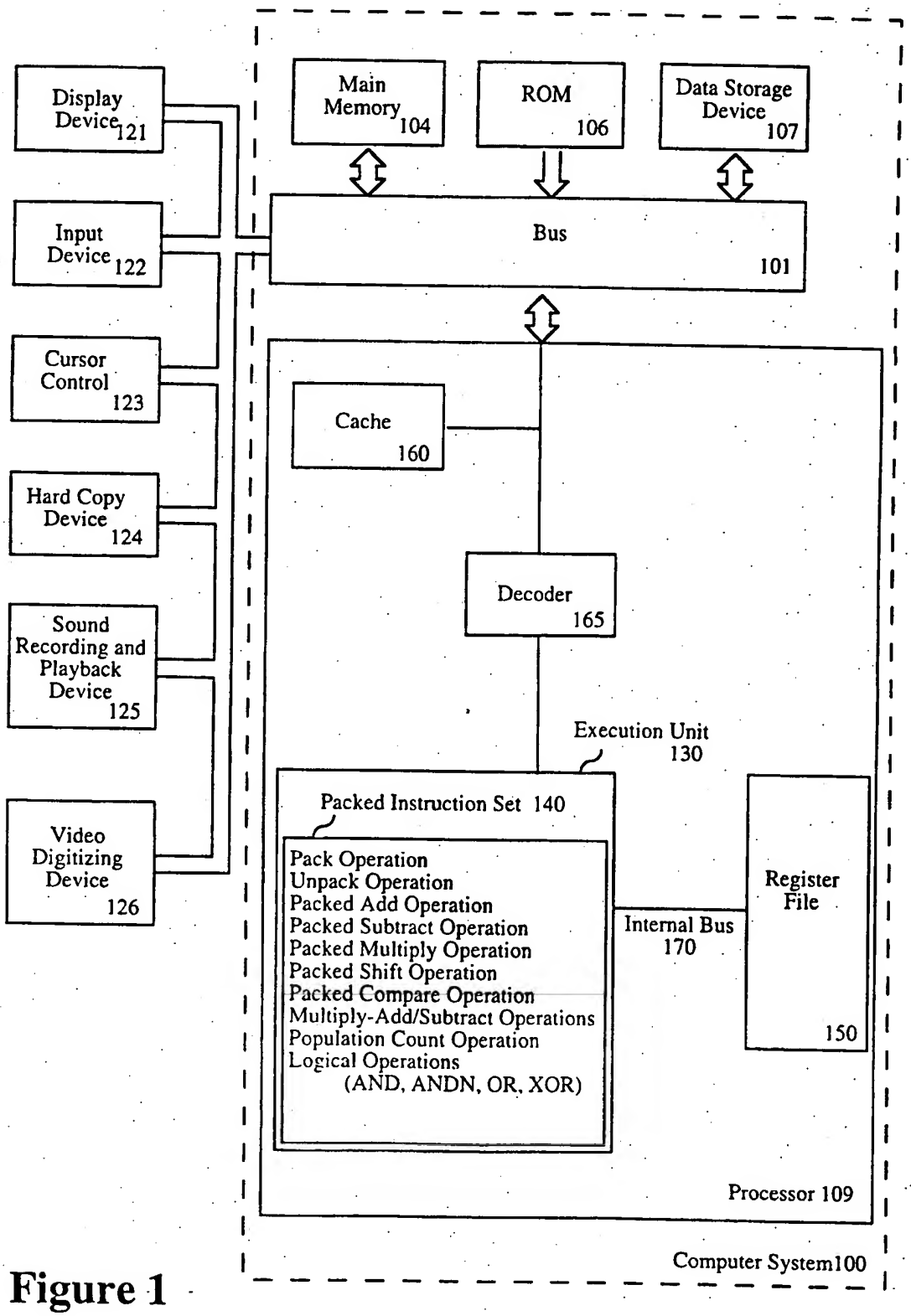

Display Device 121
Input Device 122
Cursor Control 123
Hard Copy Device 124
Sound Recording and Playback Device 125
Video Digitizing Device 126
Main Memory 104
ROM 106
Data Storage Device 107
Bus 101
Cache 160
Decoder 165
Execution Unit 130
Packed Instruction Set 140
Pack Operation
Unpack Operation
Packed Add Operation
Packed Subtract Operation
Packed Multiply Operation
Packed Shift Operation
Packed Compare Operation
Multiply-Add/Subtract Operations
Population Count Operation
Logical Operations
(AND, ANDN, OR, XOR)
Internal Bus 170
Register File 150
Processor 109
Computer System100

Figure 1

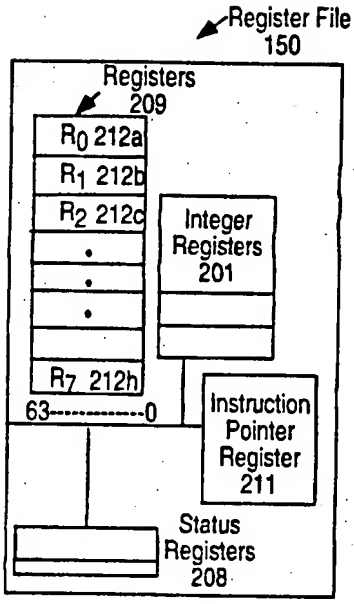


Figure 2

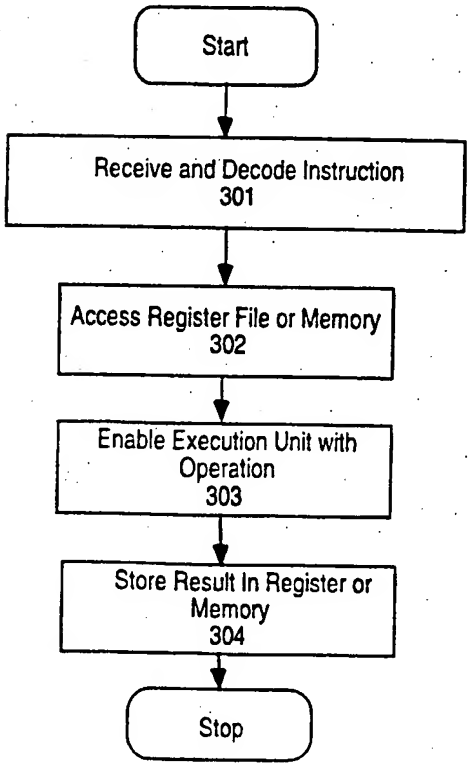


Figure 3

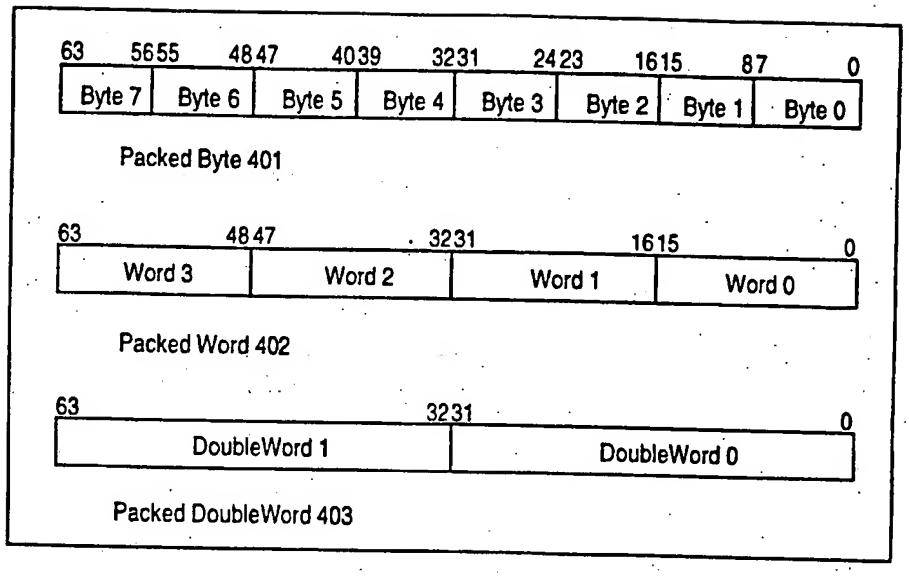


Figure 4

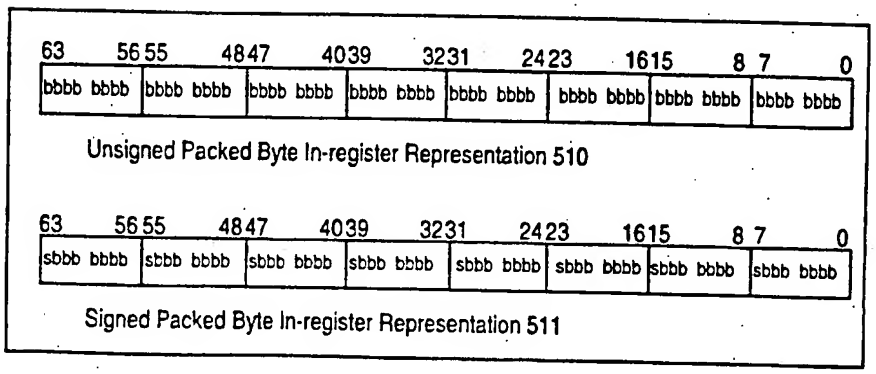


Figure 5a

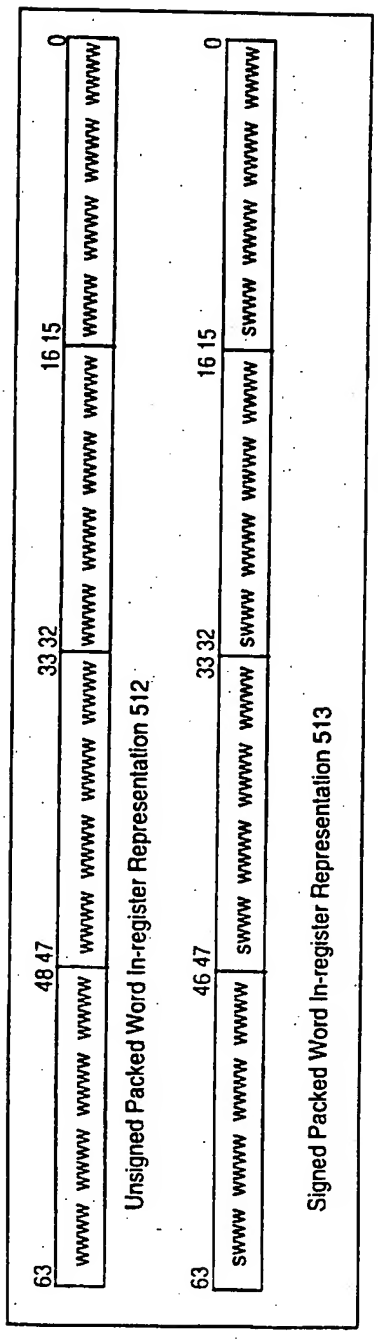


Figure 5b

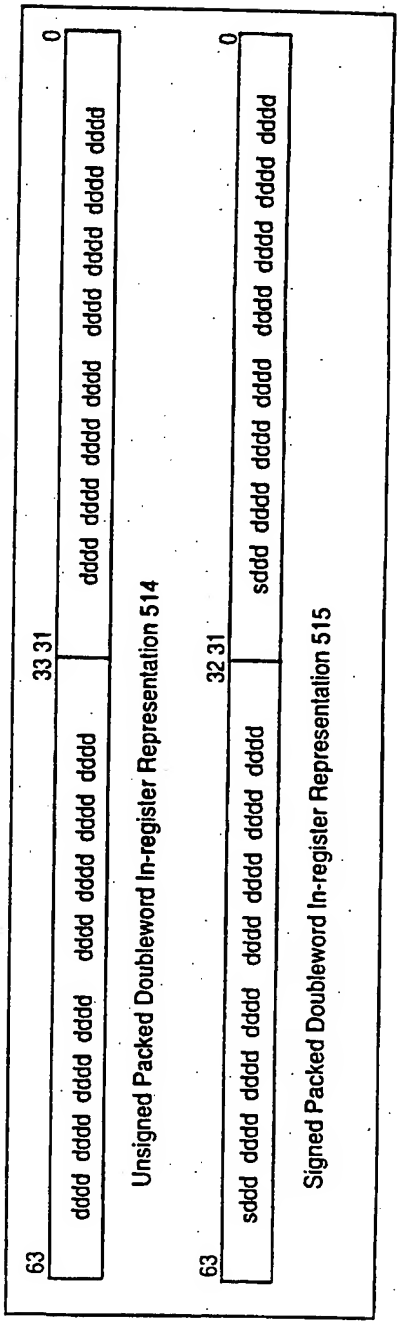


Figure 5c

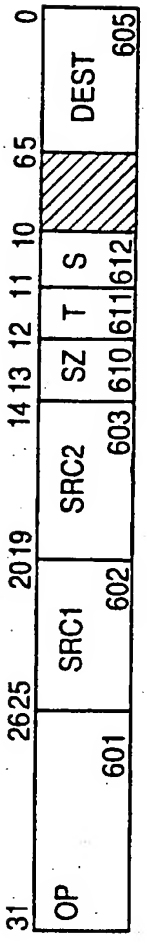


Figure 6a

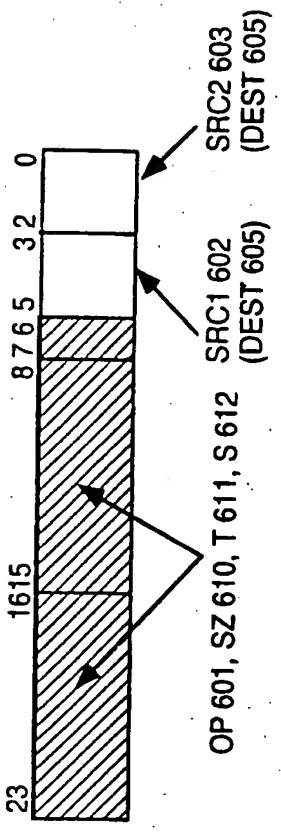


Figure 6b

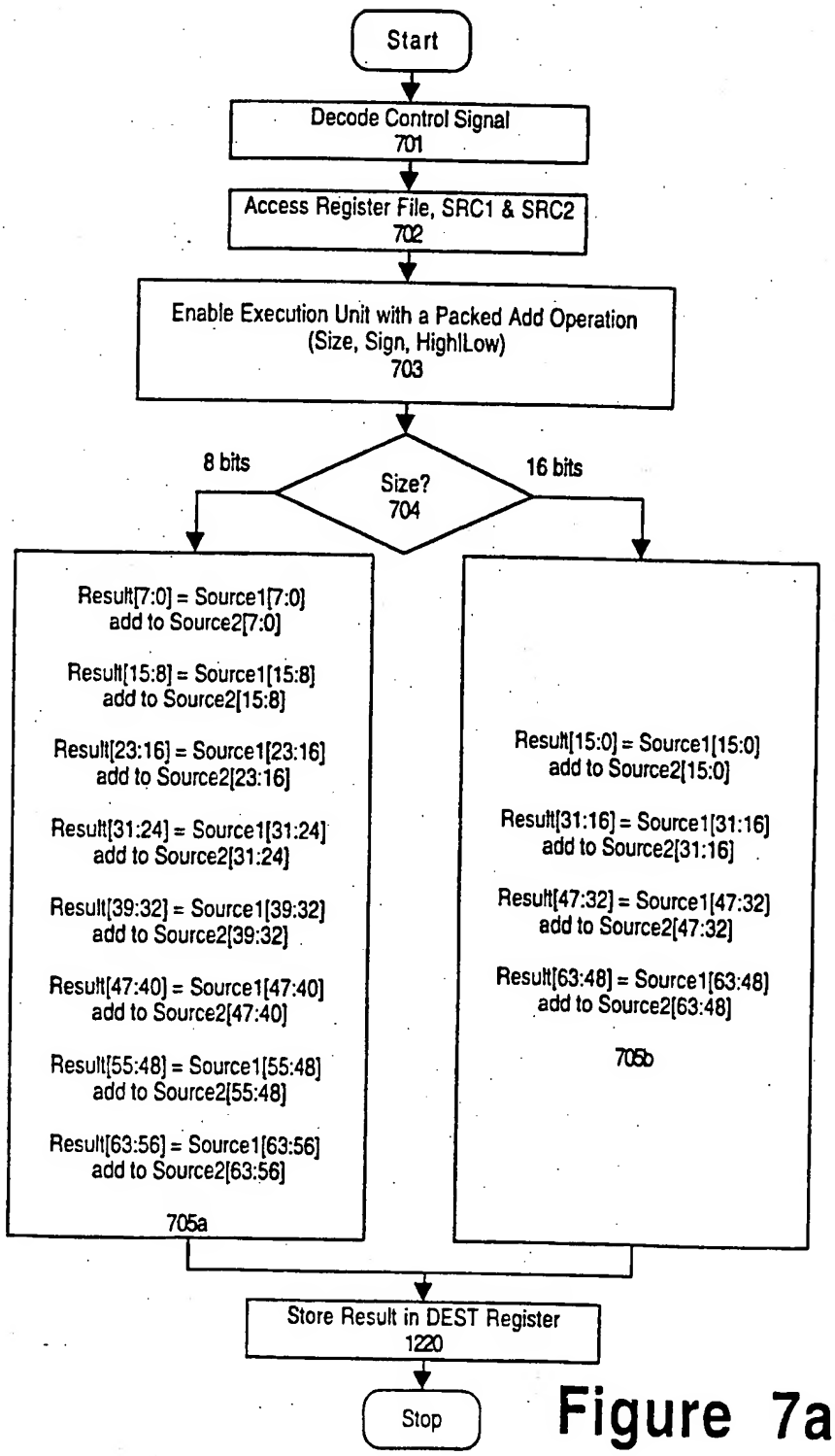

Start
Decode Control Signal
701
Access Register File, SRC1 & SRC2
702
Enable Execution Unit with a Packed Add Operation
(Size, Sign, HighILow)
703
Size?
704
8 bits
16 bits
Result[7:0] = Source1[7:0]
add to Source2[7:0]
Result[15:8] = Source1[15:8]
add to Source2[15:8]
Result[23:16] = Source1[23:16]
add to Source2[23:16]
Result[31:24] = Source1[31:24]
add to Source2[31:24]
Result[39:32] = Source1[39:32]
add to Source2[39:32]
Result[47:40] = Source1[47:40]
add to Source2[47:40]
Result[55:48] = Source1[55:48]
add to Source2[55:48]
Result[63:56] = Source1[63:56]
add to Source2[63:56]
705a
Result[15:0] = Source1[15:0]
add to Source2[15:0]
Result[31:16] = Source1[31:16]
add to Source2[31:16]
Result[47:32] = Source1[47:32]
add to Source2[47:32]
Result[63:48] = Source1[63:48]
add to Source2[63:48]
705b
Store Result in DEST Register
1220
Stop

Figure 7a

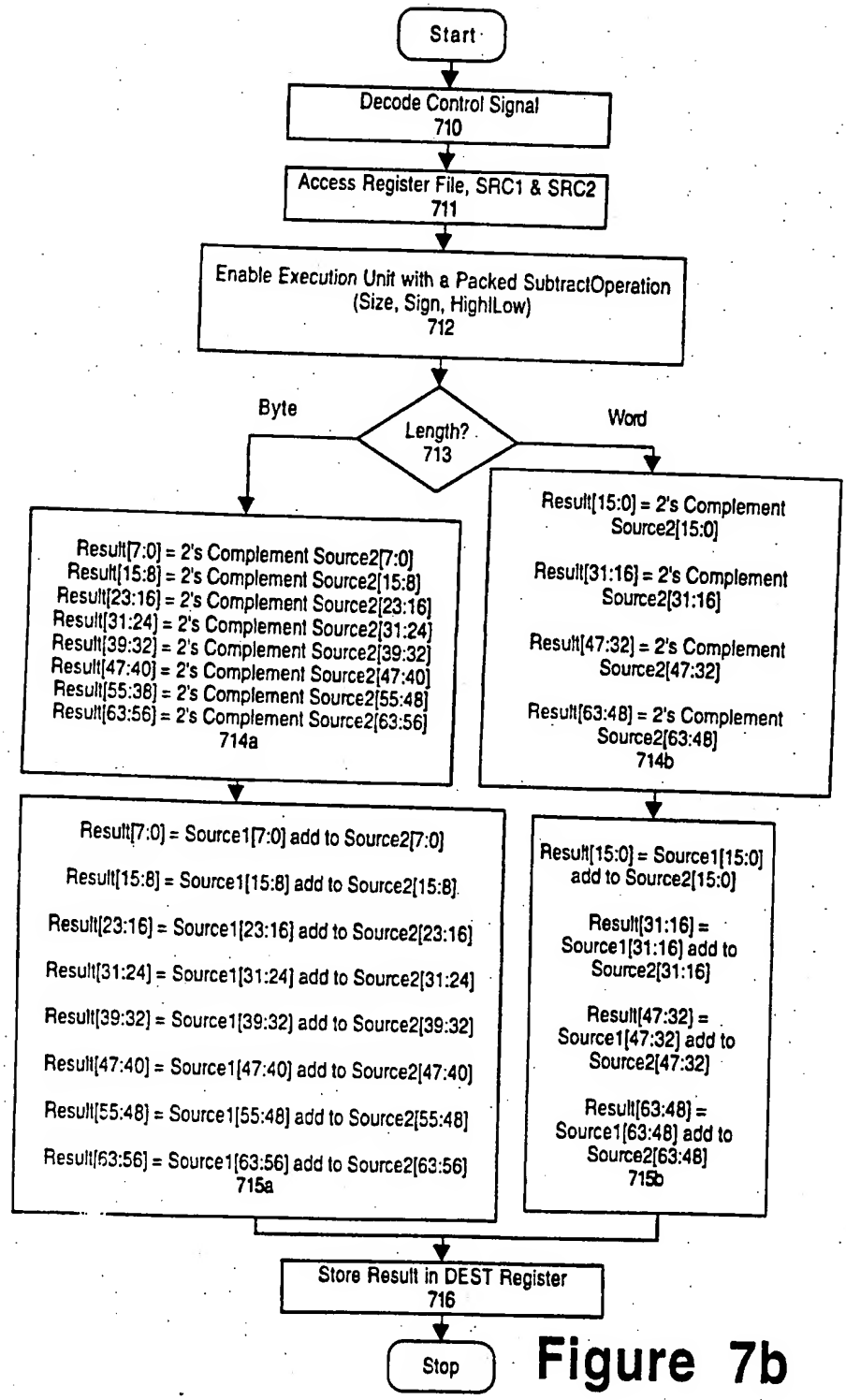

Start
Decode Control Signal
710
Access Register File, SRC1 & SRC2
711
Enable Execution Unit with a Packed SubtractOperation
(Size, Sign, HighLow)
712
Length?
713
Byte
Word
Result[7:0] = 2's Complement Source2[7:0]
Result[15:8] = 2's Complement Source2[15:8]
Result[23:16] = 2's Complement Source2[23:16]
Result[31:24] = 2's Complement Source2[31:24]
Result[39:32] = 2's Complement Source2[39:32]
Result[47:40] = 2's Complement Source2[47:40]
Result[55:38] = 2's Complement Source2[55:48]
Result[63:56] = 2's Complement Source2[63:56]
714a
Result[15:0] = 2's Complement Source2[15:0]
Result[31:16] = 2's Complement Source2[31:16]
Result[47:32] = 2's Complement Source2[47:32]
Result[63:48] = 2's Complement Source2[63:48]
714b
Result[7:0] = Source1[7:0] add to Source2[7:0]
Result[15:8] = Source1[15:8] add to Source2[15:8]
Result[23:16] = Source1[23:16] add to Source2[23:16]
Result[31:24] = Source1[31:24] add to Source2[31:24]
Result[39:32] = Source1[39:32] add to Source2[39:32]
Result[47:40] = Source1[47:40] add to Source2[47:40]
Result[55:48] = Source1[55:48] add to Source2[55:48]
Result[63:56] = Source1[63:56] add to Source2[63:56]
715a
Result[15:0] = Source1[15:0] add to Source2[15:0]
Result[31:16] = Source1[31:16] add to Source2[31:16]
Result[47:32] = Source1[47:32] add to Source2[47:32]
Result[63:48] = Source1[63:48] add to Source2[63:48]
715b
Store Result in DEST Register
716
Stop

Figure 7b

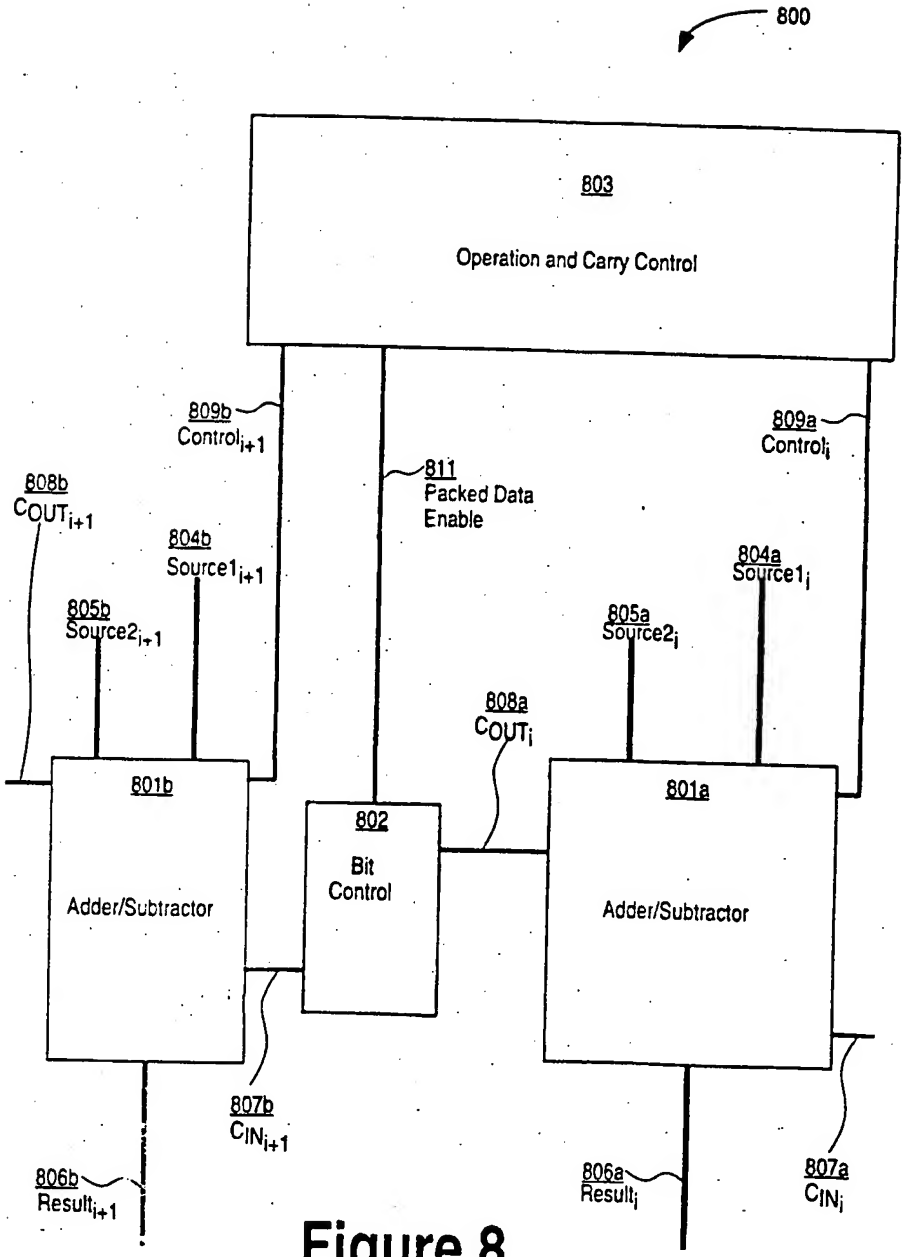

800
803
Operation and Carry Control
809b
Control$_{i+1}$
809a
Control$_i$
808b
COUT$_{i+1}$
811
Packed Data
Enable
804b
Source1$_{i+1}$
804a
Source1$_i$
805b
Source2$_{i+1}$
805a
Source2$_i$
808a
COUT$_i$
801b
801a
802
Bit
Control
Adder/Subtractor
Adder/Subtractor
807b
CIN$_{i+1}$
806b
Result$_{i+1}$
806a
Result$_i$
807a
CIN$_i$

Figure 8

Figure 9

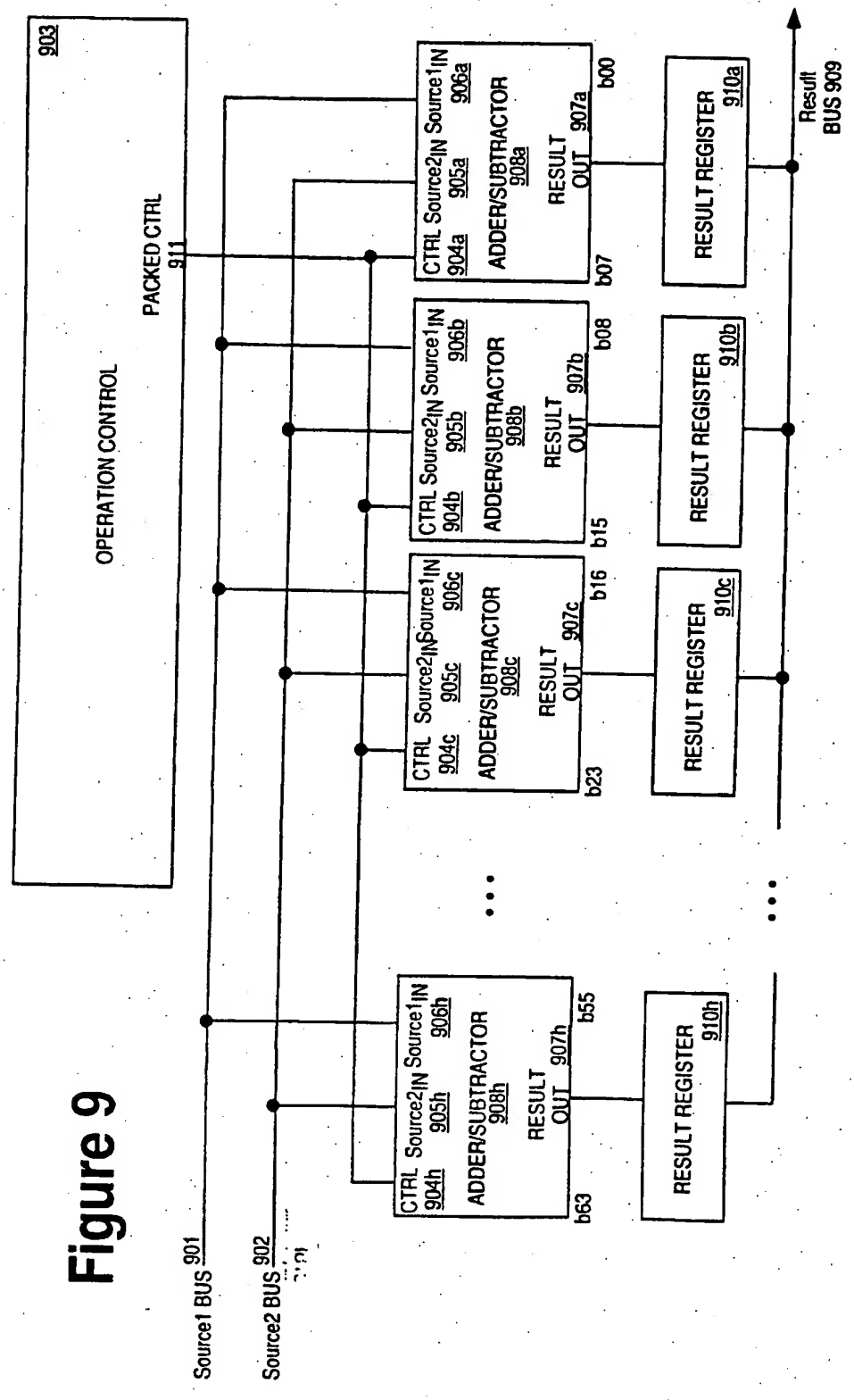

903
OPERATION CONTROL
PACKED CTRL
911
Source1 BUS 901
Source2 BUS 902
CTRL 904a
Source2IN 905a
Source1 IN 906a
ADDER/SUBTRACTOR 908a
RESULT OUT 907a
b00
b07
RESULT REGISTER 910a
CTRL 904b
Source2IN 905b
Source1 IN 906b
ADDER/SUBTRACTOR 908b
RESULT OUT 907b
b08
b15
RESULT REGISTER 910b
CTRL 904c
Source2IN 905c
Source1 IN 906c
ADDER/SUBTRACTOR 908c
RESULT OUT 907c
b16
b23
RESULT REGISTER 910c
CTRL 904h
Source2IN 905h
Source1 IN 906h
ADDER/SUBTRACTOR 908h
RESULT OUT 907h
b55
b63
RESULT REGISTER 910h
Result BUS 909

Figure 10

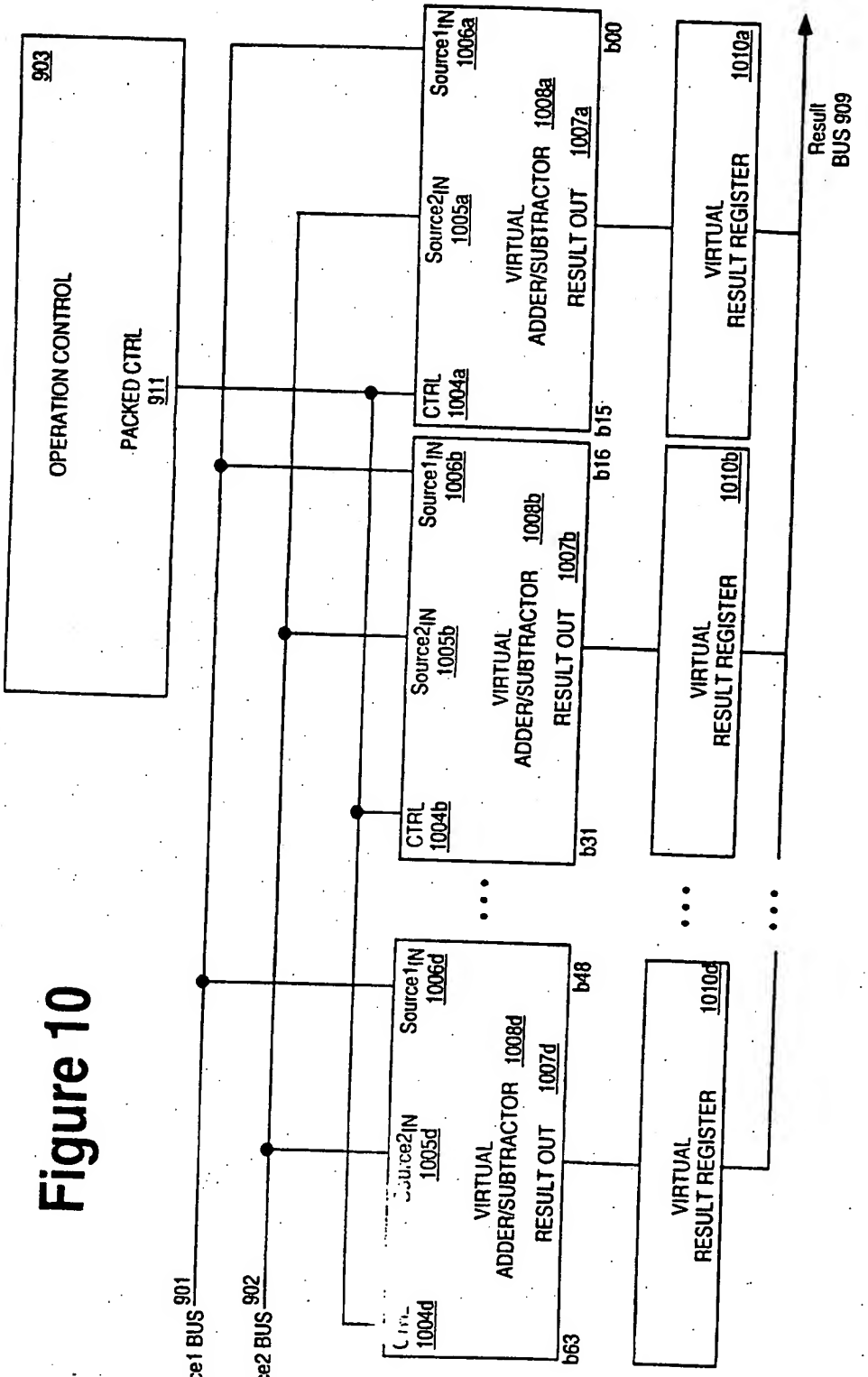
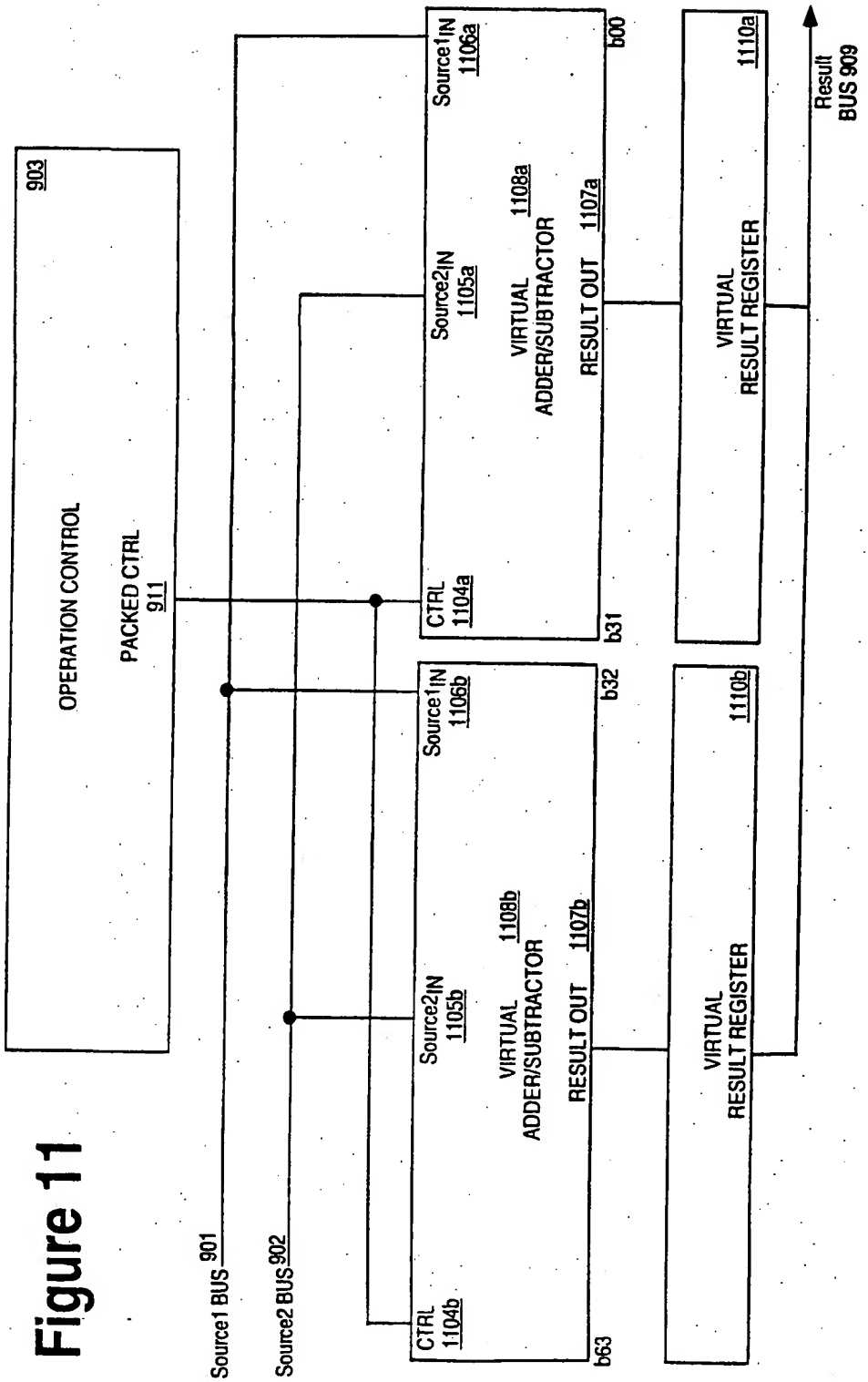

OPERATION CONTROL
903
PACKED CTRL
911
Source1 BUS 901
Source2 BUS 902
CTRL 1004a
Source2IN 1005a
Source1IN 1006a
VIRTUAL ADDER/SUBTRACTOR 1008a
RESULT OUT 1007a
b00
b15
CTRL 1004b
Source2IN 1005b
Source1IN 1006b
VIRTUAL ADDER/SUBTRACTOR 1008b
RESULT OUT 1007b
b16
b31
CTRL 1004d
Source2IN 1005d
Source1IN 1006d
VIRTUAL ADDER/SUBTRACTOR 1008d
RESULT OUT 1007d
b48
b63
VIRTUAL RESULT REGISTER 1010a
VIRTUAL RESULT REGISTER 1010b
VIRTUAL RESULT REGISTER 1010d
Result BUS 909

Figure 11


OPERATION CONTROL
903
PACKED CTRL
911
Source1 BUS 901
Source2 BUS 902
CTRL
1104a
Source1IN
1106a
Source2IN
1105a
VIRTUAL
ADDER/SUBTRACTOR
1108a
RESULT OUT
1107a
b00
b31
CTRL
1104b
Source1IN
1106b
Source2IN
1105b
VIRTUAL
ADDER/SUBTRACTOR
1108b
RESULT OUT
1107b
b32
b63
VIRTUAL
RESULT REGISTER
1110a
VIRTUAL
RESULT REGISTER
1110b
Result
BUS 909

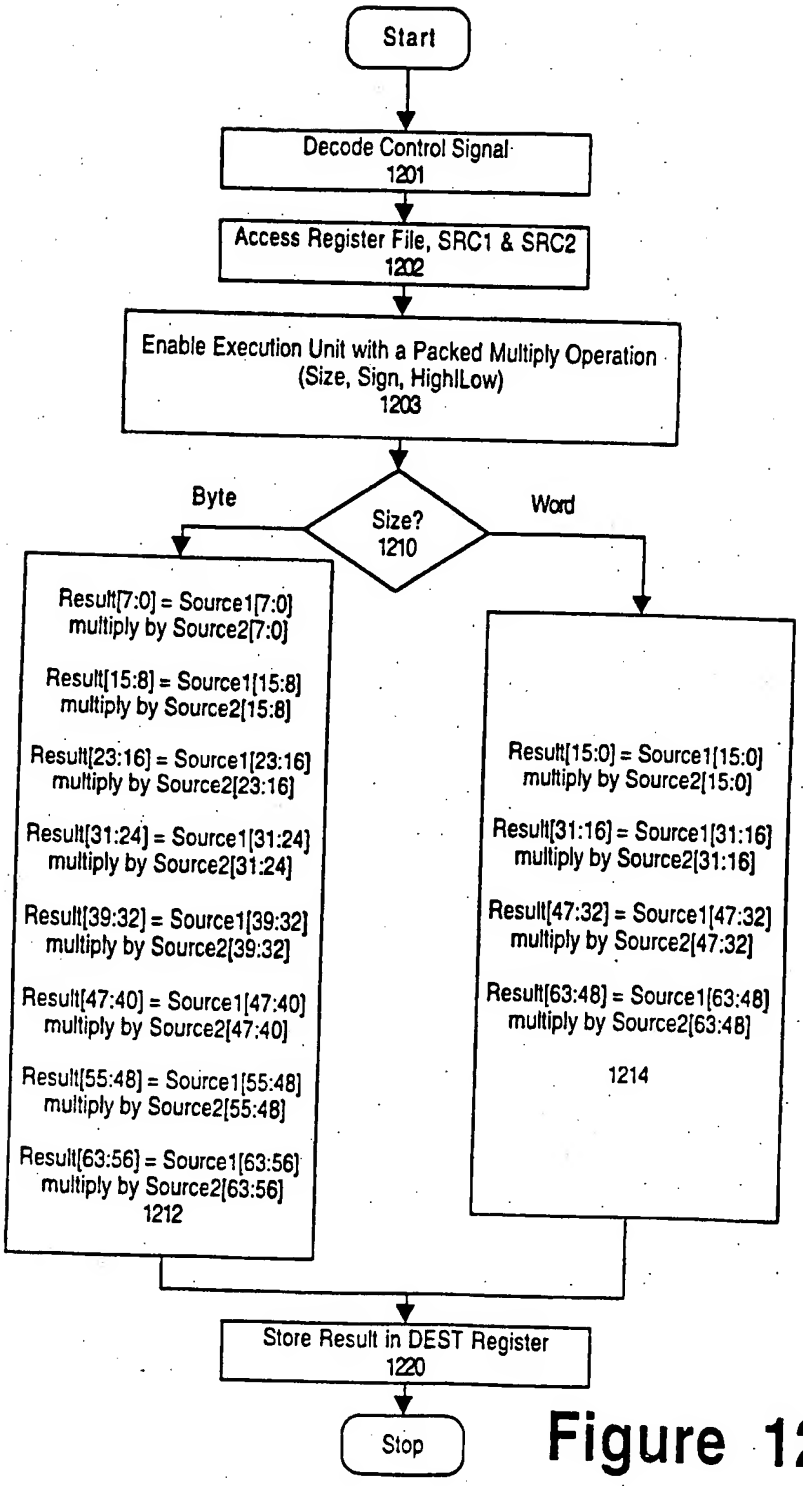

Start
Decode Control Signal
1201
Access Register File, SRC1 & SRC2
1202
Enable Execution Unit with a Packed Multiply Operation
(Size, Sign, HighlLow)
1203
Size?
1210
Byte
Word
Result[7:0] = Source1[7:0]
multiply by Source2[7:0]
Result[15:8] = Source1[15:8]
multiply by Source2[15:8]
Result[23:16] = Source1[23:16]
multiply by Source2[23:16]
Result[31:24] = Source1[31:24]
multiply by Source2[31:24]
Result[39:32] = Source1[39:32]
multiply by Source2[39:32]
Result[47:40] = Source1[47:40]
multiply by Source2[47:40]
Result[55:48] = Source1[55:48]
multiply by Source2[55:48]
Result[63:56] = Source1[63:56]
multiply by Source2[63:56]
1212
Result[15:0] = Source1[15:0]
multiply by Source2[15:0]
Result[31:16] = Source1[31:16]
multiply by Source2[31:16]
Result[47:32] = Source1[47:32]
multiply by Source2[47:32]
Result[63:48] = Source1[63:48]
multiply by Source2[63:48]
1214
Store Result in DEST Register
1220
Stop

Figure 12

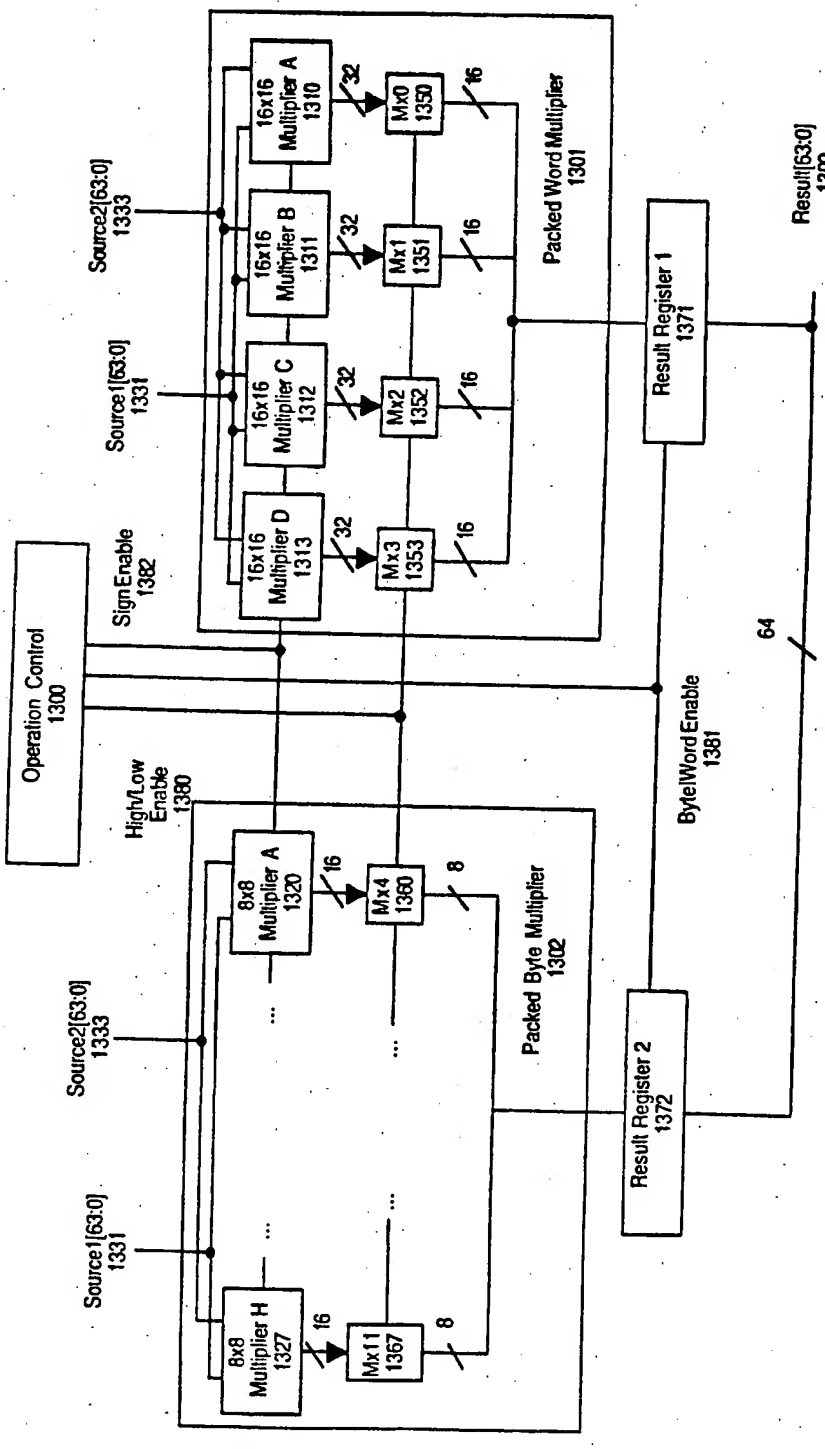

Operation Control 1300
Source2[63:0] 1333
Source1[63:0] 1331
SignEnable 1382
High/Low Enable 1380
Byte/Word Enable 1381
16x16 Multiplier A 1310
16x16 Multiplier B 1311
16x16 Multiplier C 1312
16x16 Multiplier D 1313
Mx0 1350
Mx1 1351
Mx2 1352
Mx3 1353
Packed Word Multiplier 1301
8x8 Multiplier A 1320
8x8 Multiplier H 1327
Mx4 1360
Mx11 1367
Packed Byte Multiplier 1302
Result Register 1 1371
Result Register 2 1372
Result[63:0] 1390
32
16
8
64

Figure 13

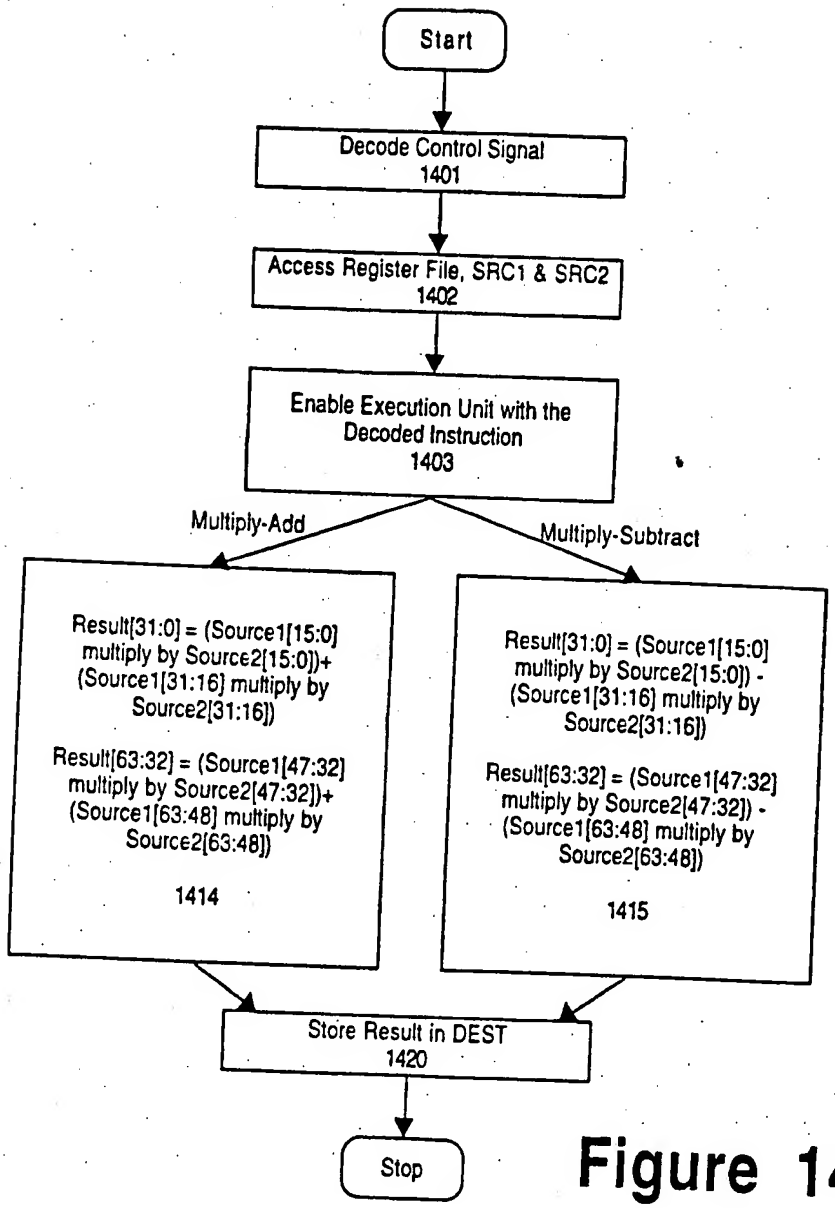

Start
Decode Control Signal
1401
Access Register File, SRC1 & SRC2
1402
Enable Execution Unit with the Decoded Instruction
1403
Multiply-Add
Multiply-Subtract
Result[31:0] = (Source1[15:0] multiply by Source2[15:0])+ (Source1[31:16] multiply by Source2[31:16])
Result[63:32] = (Source1[47:32] multiply by Source2[47:32])+ (Source1[63:48] multiply by Source2[63:48])
1414
Result[31:0] = (Source1[15:0] multiply by Source2[15:0]) - (Source1[31:16] multiply by Source2[31:16])
Result[63:32] = (Source1[47:32] multiply by Source2[47:32]) - (Source1[63:48] multiply by Source2[63:48])
1415
Store Result in DEST
1420
Stop

Figure 14

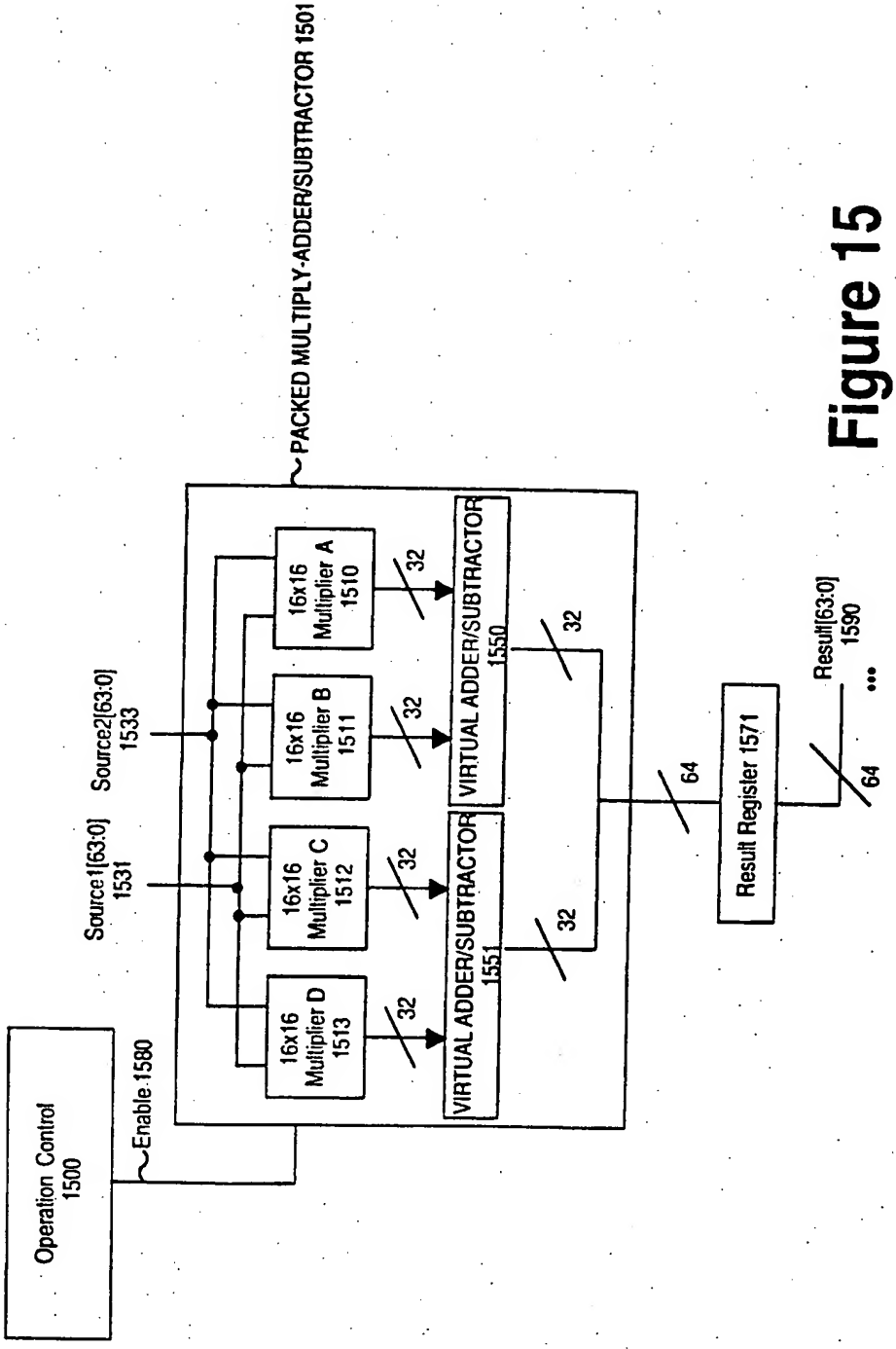

PACKED MULTIPLY-ADDER/SUBTRACTOR 1501
Operation Control 1500
Enable 1580
Source1[63:0] 1531
Source2[63:0] 1533
16x16 Multiplier A 1510
16x16 Multiplier B 1511
16x16 Multiplier C 1512
16x16 Multiplier D 1513
32
VIRTUAL ADDER/SUBTRACTOR 1550
VIRTUAL ADDER/SUBTRACTOR 1551
64
Result Register 1571
Result[63:0] 1590

Figure 15

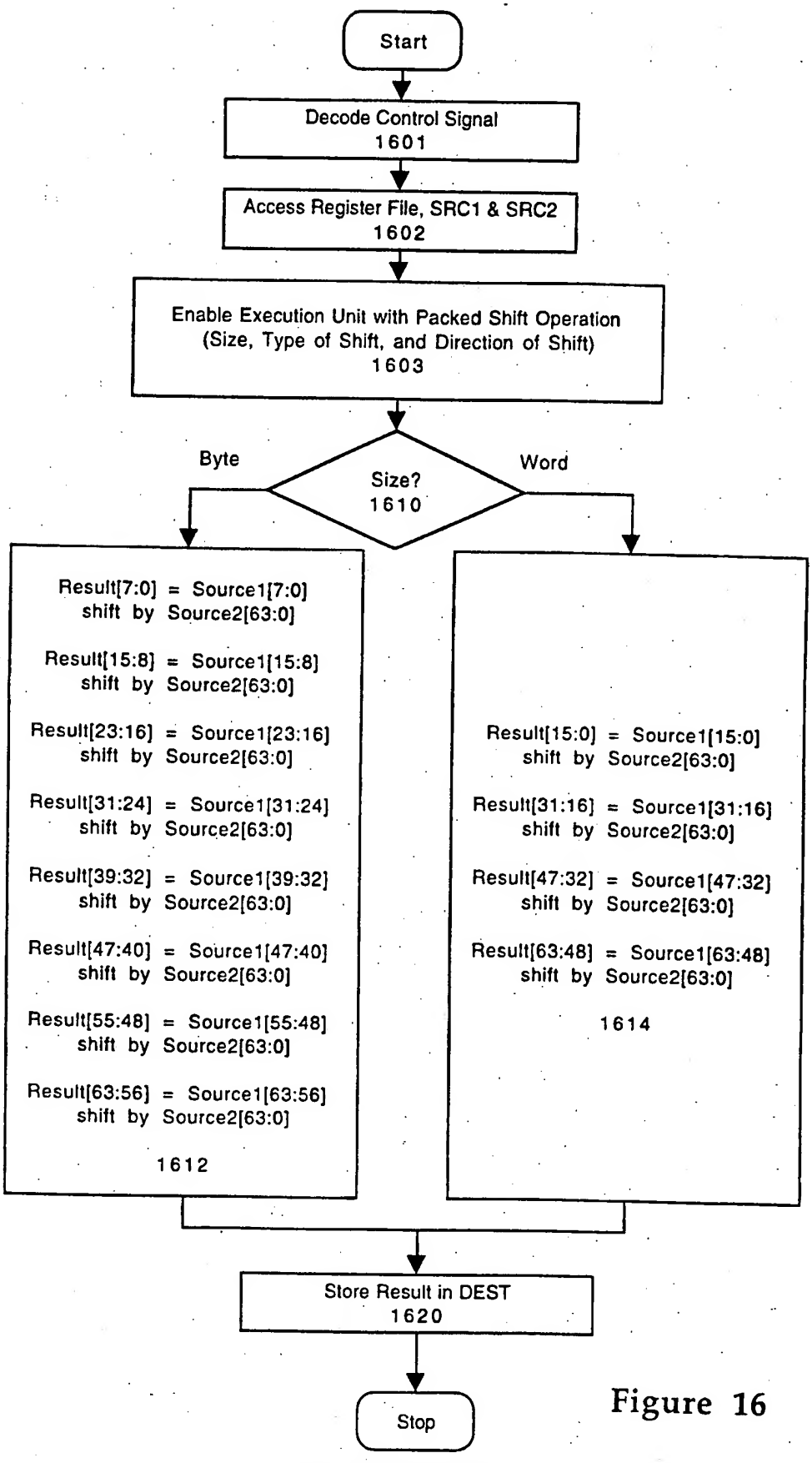
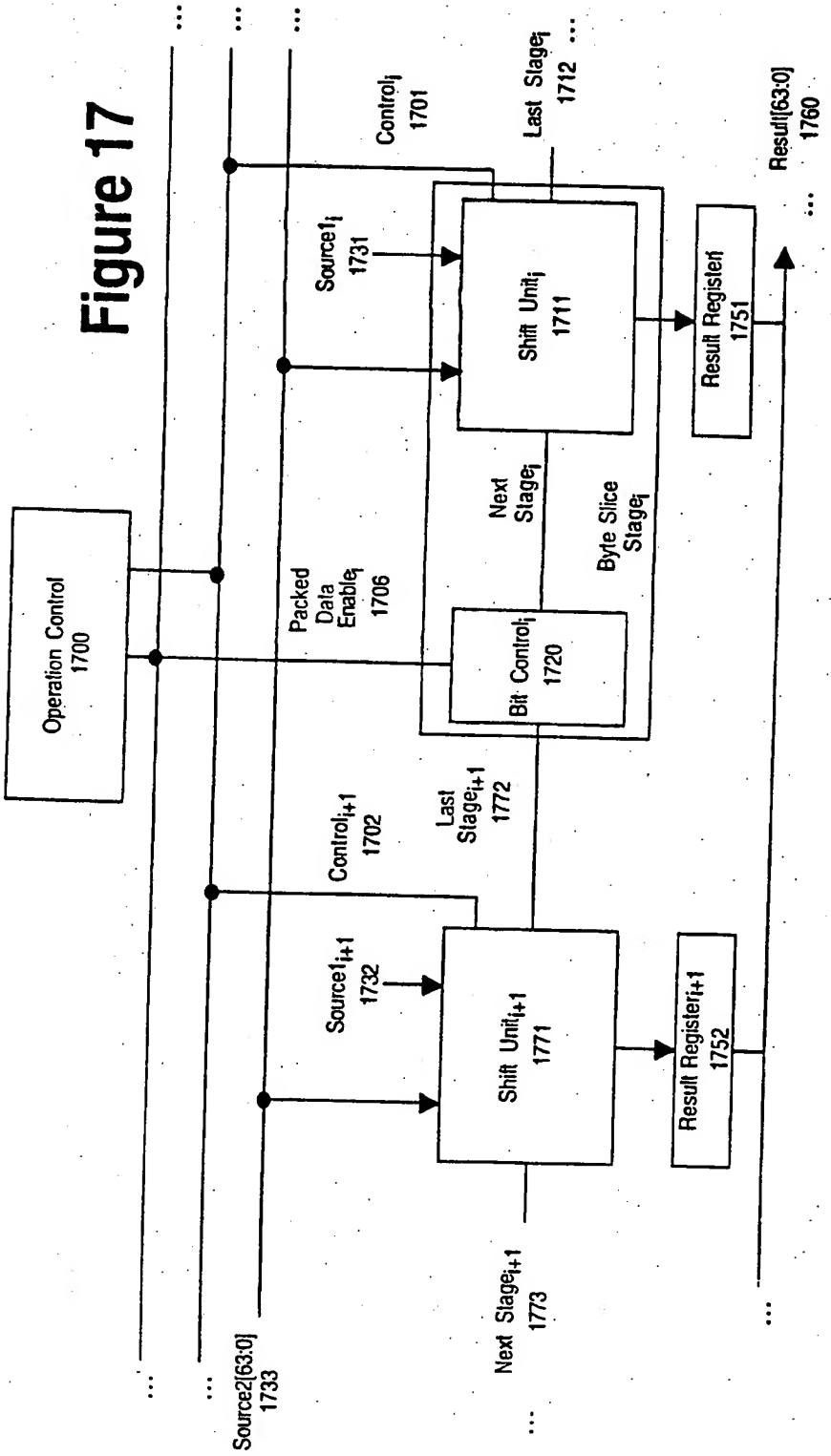

Start
Decode Control Signal
1601
Access Register File, SRC1 & SRC2
1602
Enable Execution Unit with Packed Shift Operation
(Size, Type of Shift, and Direction of Shift)
1603
Byte
Size?
1610
Word
Result[7:0] = Source1[7:0]
shift by Source2[63:0]
Result[15:8] = Source1[15:8]
shift by Source2[63:0]
Result[23:16] = Source1[23:16]
shift by Source2[63:0]
Result[31:24] = Source1[31:24]
shift by Source2[63:0]
Result[39:32] = Source1[39:32]
shift by Source2[63:0]
Result[47:40] = Source1[47:40]
shift by Source2[63:0]
Result[55:48] = Source1[55:48]
shift by Source2[63:0]
Result[63:56] = Source1[63:56]
shift by Source2[63:0]
1612
Result[15:0] = Source1[15:0]
shift by Source2[63:0]
Result[31:16] = Source1[31:16]
shift by Source2[63:0]
Result[47:32] = Source1[47:32]
shift by Source2[63:0]
Result[63:48] = Source1[63:48]
shift by Source2[63:0]
1614
Store Result in DEST
1620
Stop

Figure 16

Figure 17


Operation Control 1700
Control$_i$ 1701
Source1$_i$ 1731
Packed Data Enable$_i$ 1706
Shift Unit$_i$ 1711
Last Stage$_i$ 1712
Result Register 1751
Result[63:0] 1760
Next Stage$_i$
Byte Slice Stage$_i$
Bit Control$_i$ 1720
Control$_{i+1}$ 1702
Last Stage$_{i+1}$ 1772
Source1$_{i+1}$ 1732
Shift Unit$_{i+1}$ 1771
Result Register$_{i+1}$ 1752
Next Stage$_{i+1}$ 1773
Source2[63:0] 1733

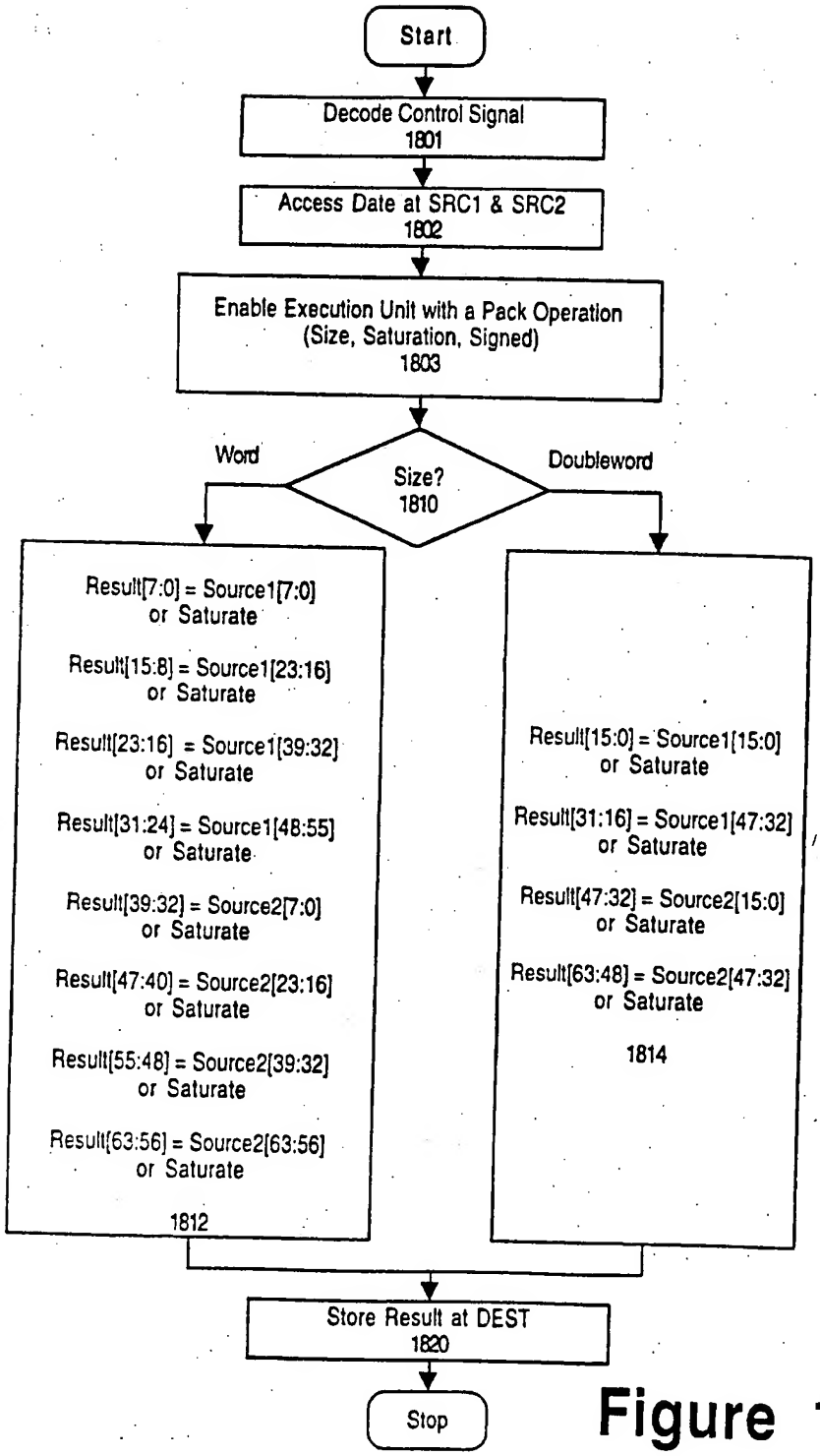

Start
Decode Control Signal
1801
Access Date at SRC1 & SRC2
1802
Enable Execution Unit with a Pack Operation
(Size, Saturation, Signed)
1803
Word
Size?
1810
Doubleword
Result[7:0] = Source1[7:0]
or Saturate
Result[15:8] = Source1[23:16]
or Saturate
Result[23:16] = Source1[39:32]
or Saturate
Result[31:24] = Source1[48:55]
or Saturate
Result[39:32] = Source2[7:0]
or Saturate
Result[47:40] = Source2[23:16]
or Saturate
Result[55:48] = Source2[39:32]
or Saturate
Result[63:56] = Source2[63:56]
or Saturate
1812
Result[15:0] = Source1[15:0]
or Saturate
Result[31:16] = Source1[47:32]
or Saturate
Result[47:32] = Source2[15:0]
or Saturate
Result[63:48] = Source2[47:32]
or Saturate
1814
Store Result at DEST
1820
Stop

Figure 18

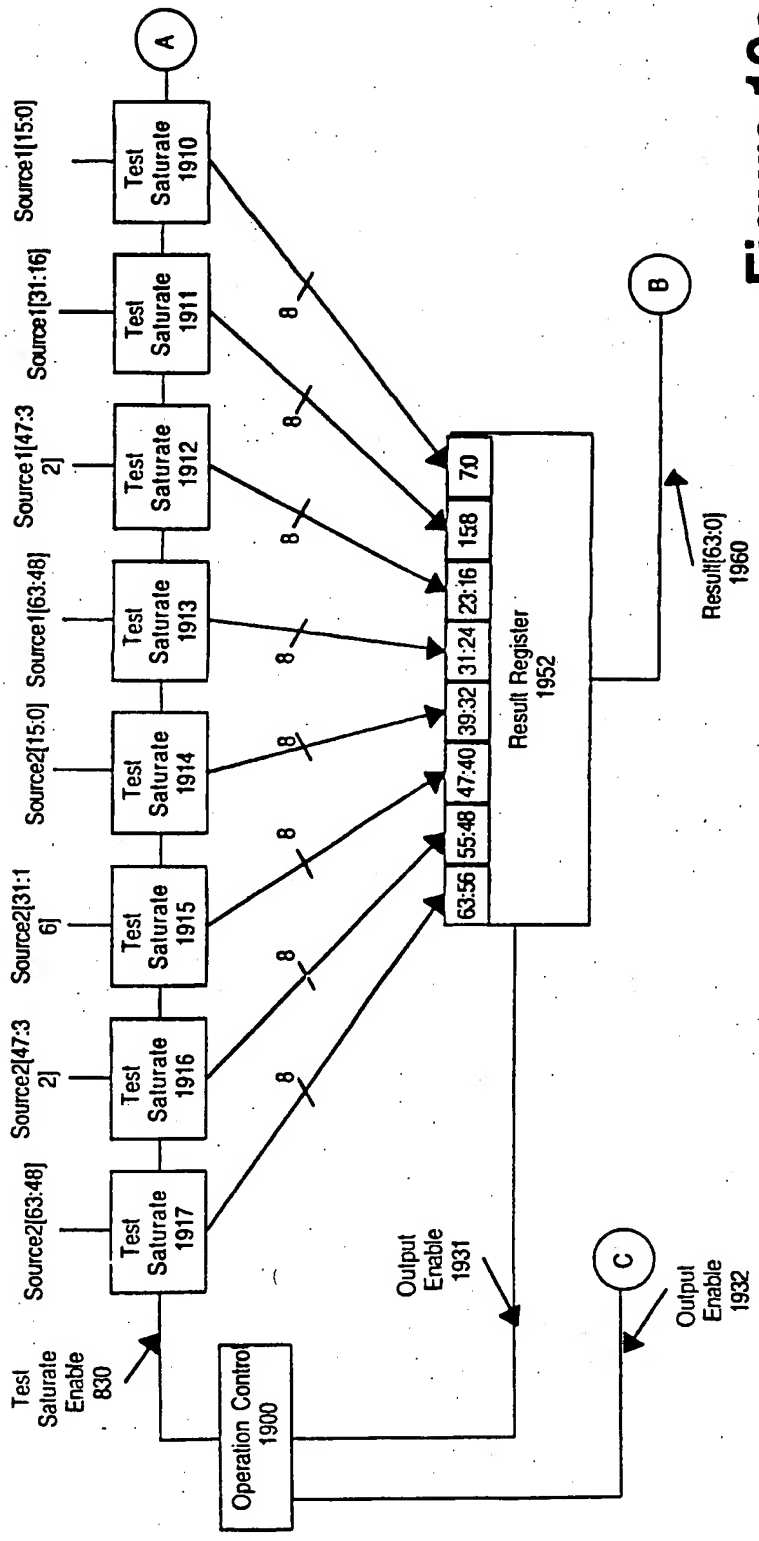

A
B
C
Source1[15:0]
Source1[31:16]
Source1[47:32]
Source1[63:48]
Source2[15:0]
Source2[31:16]
Source2[47:32]
Source2[63:48]
Test Saturate 1910
Test Saturate 1911
Test Saturate 1912
Test Saturate 1913
Test Saturate 1914
Test Saturate 1915
Test Saturate 1916
Test Saturate 1917
8
Result Register 1952
63:56
55:48
47:40
39:32
31:24
23:16
15:8
7:0
Result[63:0] 1960
Test Saturate Enable 830
Operation Control 1900
Output Enable 1931
Output Enable 1932

Figure 19a

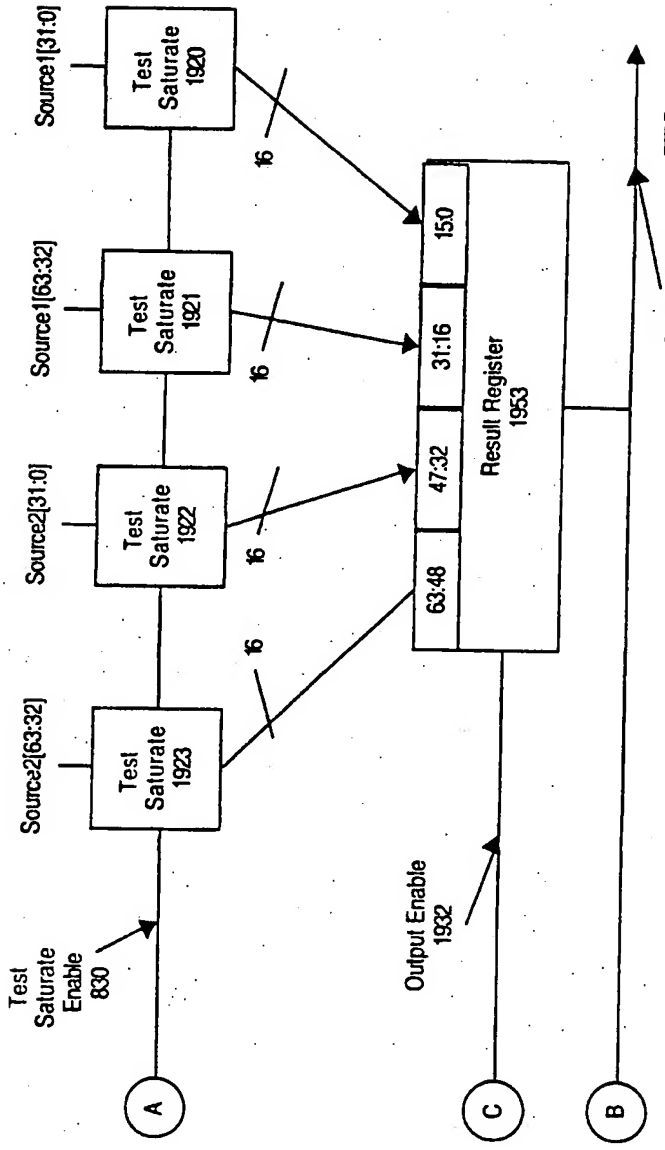

Source1[31:0]
Test Saturate 1920
Source1[63:32]
Test Saturate 1921
Source2[31:0]
Test Saturate 1922
Source2[63:32]
Test Saturate 1923
16
16
16
16
Test Saturate Enable 830
A
Output Enable 1932
C
Result Register 1953
15:0
31:16
47:32
63:48
Result[63:0] 1960
B

Figure 19b

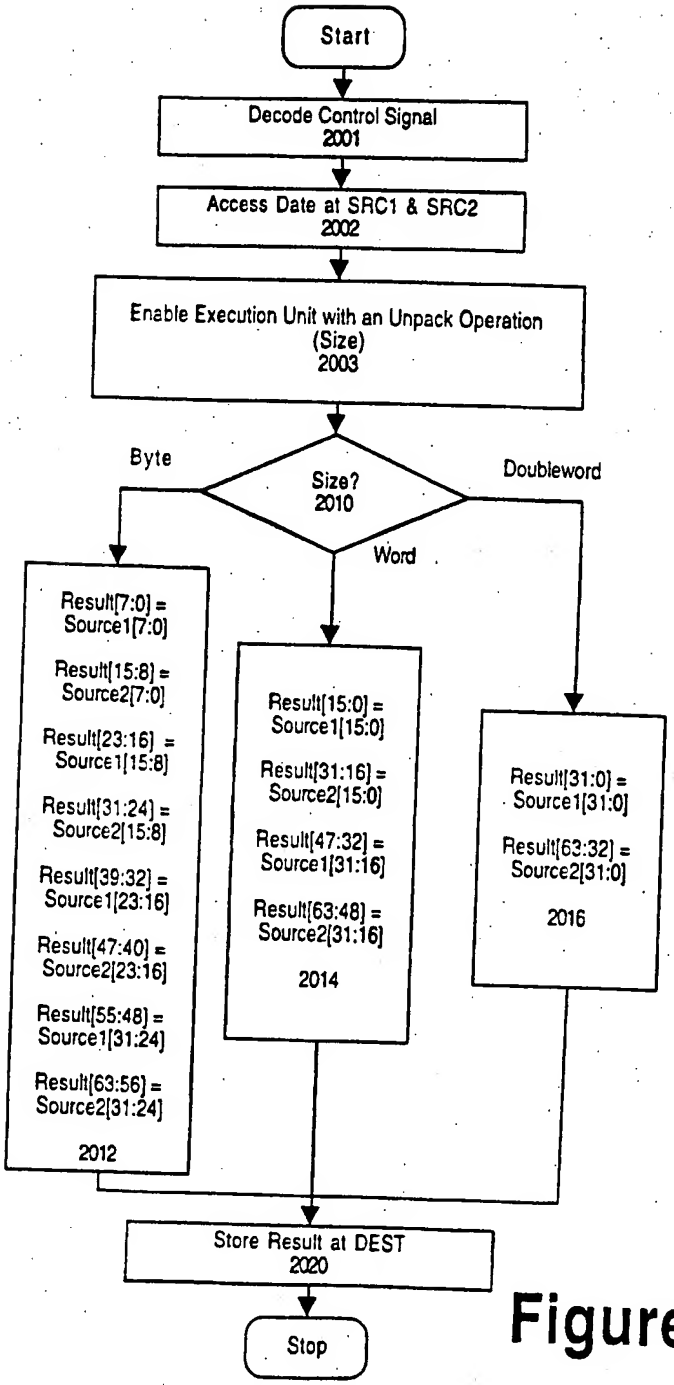
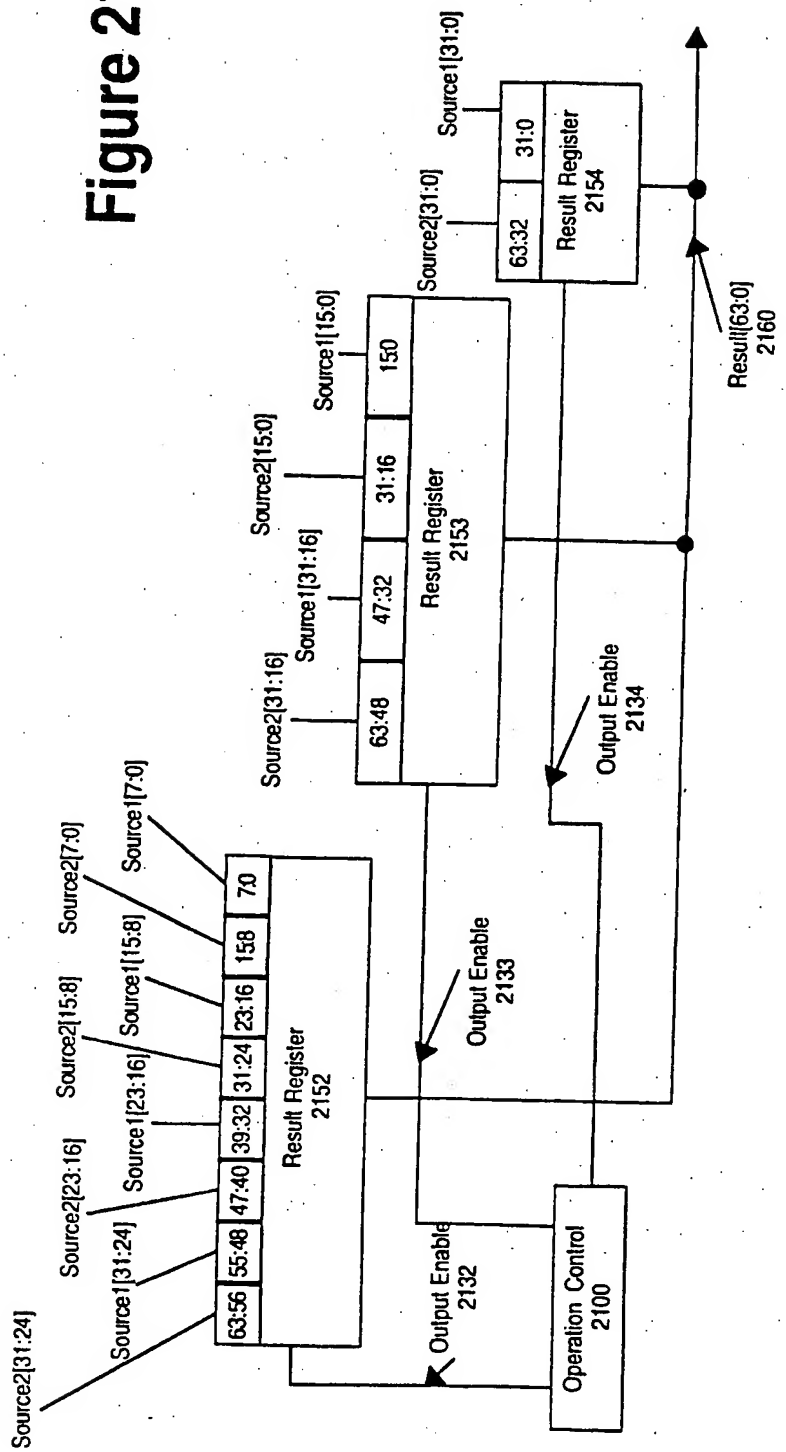

Start
Decode Control Signal
2001
Access Date at SRC1 & SRC2
2002
Enable Execution Unit with an Unpack Operation
(Size)
2003
Size?
2010
Byte
Word
Doubleword
Result[7:0] = Source1[7:0]
Result[15:8] = Source2[7:0]
Result[23:16] = Source1[15:8]
Result[31:24] = Source2[15:8]
Result[39:32] = Source1[23:16]
Result[47:40] = Source2[23:16]
Result[55:48] = Source1[31:24]
Result[63:56] = Source2[31:24]
2012
Result[15:0] = Source1[15:0]
Result[31:16] = Source2[15:0]
Result[47:32] = Source1[31:16]
Result[63:48] = Source2[31:16]
2014
Result[31:0] = Source1[31:0]
Result[63:32] = Source2[31:0]
2016
Store Result at DEST
2020
Stop

Figure 20

Figure 21


Source2[31:24]
Source1[31:24]
Source2[23:16]
Source1[23:16]
Source2[15:8]
Source1[15:8]
Source2[7:0]
Source1[7:0]
63:56
55:48
47:40
39:32
31:24
23:16
15:8
7:0
Result Register 2152
Output Enable 2132
Operation Control 2100
Source2[31:16]
Source1[31:16]
Source2[15:0]
Source1[15:0]
63:48
47:32
31:16
15:0
Result Register 2153
Output Enable 2133
Source2[31:0]
Source1[31:0]
63:32
31:0
Result Register 2154
Output Enable 2134
Result[63:0] 2160

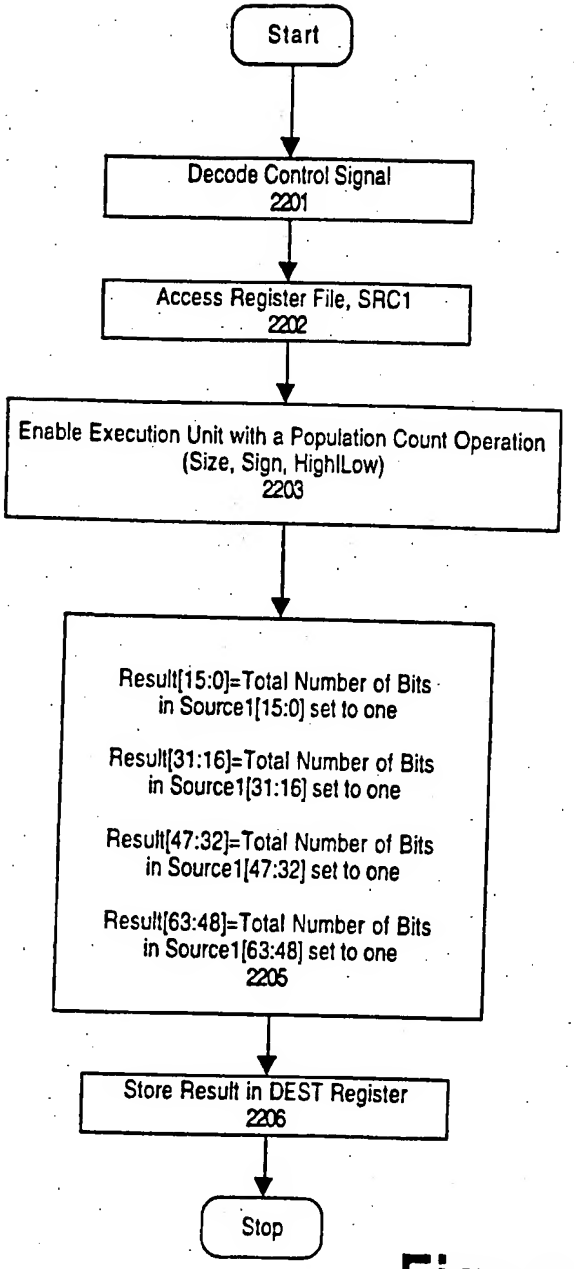

Start
Decode Control Signal
2201
Access Register File, SRC1
2202
Enable Execution Unit with a Population Count Operation
(Size, Sign, HighILow)
2203
Result[15:0]=Total Number of Bits
in Source1[15:0] set to one
Result[31:16]=Total Number of Bits
in Source1[31:16] set to one
Result[47:32]=Total Number of Bits
in Source1[47:32] set to one
Result[63:48]=Total Number of Bits
in Source1[63:48] set to one
2205
Store Result in DEST Register
2206
Stop

Figure 22

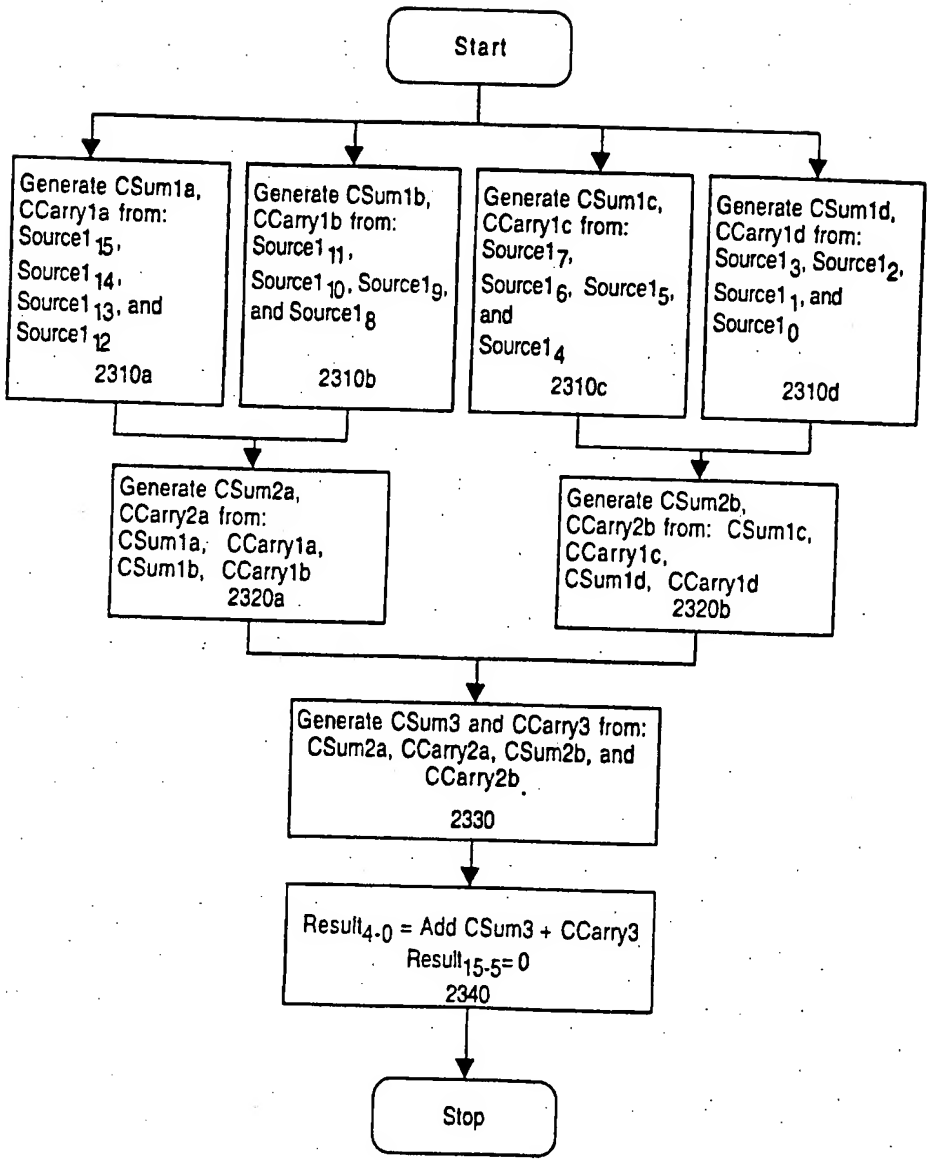

Start
Generate CSum1a, CCarry1a from: Source1 15, Source1 14, Source1 13, and Source1 12
2310a
Generate CSum1b, CCarry1b from: Source1 11, Source1 10, Source1 9, and Source1 8
2310b
Generate CSum1c, CCarry1c from: Source1 7, Source1 6, Source1 5, and Source1 4
2310c
Generate CSum1d, CCarry1d from: Source1 3, Source1 2, Source1 1, and Source1 0
2310d
Generate CSum2a, CCarry2a from: CSum1a, CCarry1a, CSum1b, CCarry1b
2320a
Generate CSum2b, CCarry2b from: CSum1c, CCarry1c, CSum1d, CCarry1d
2320b
Generate CSum3 and CCarry3 from: CSum2a, CCarry2a, CSum2b, and CCarry2b
2330
Result 4-0 = Add CSum3 + CCarry3
Result 15-5 = 0
2340
Stop

Figure 23

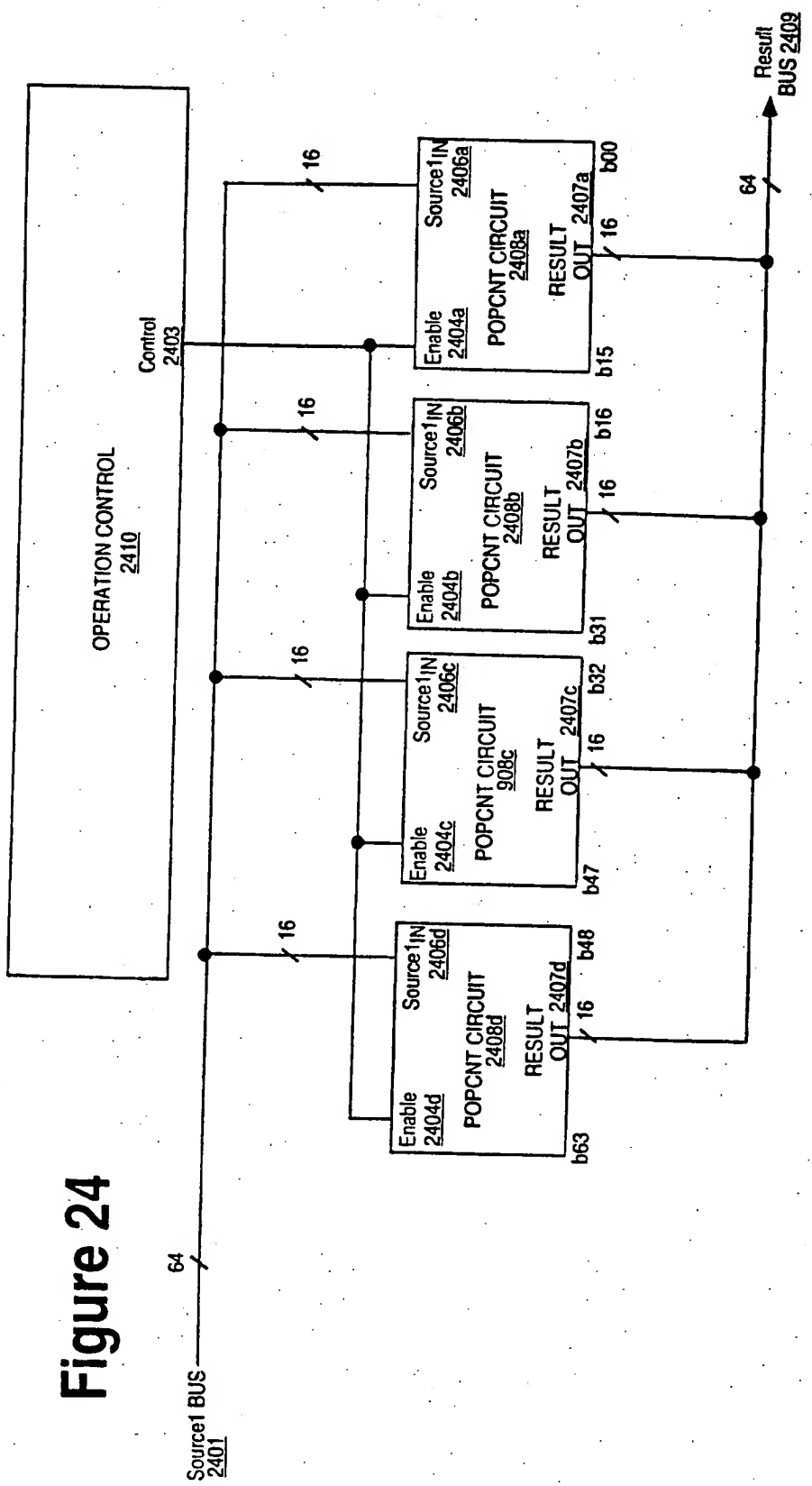

OPERATION CONTROL
2410
Control
2403
Source1 BUS
2401
64
16
Source1IN
2406a
Enable
2404a
POPCNT CIRCUIT
2408a
RESULT
OUT 2407a
b15
b00
Source1IN
2406b
Enable
2404b
POPCNT CIRCUIT
2408b
RESULT
OUT 2407b
b31
b16
Source1IN
2406c
Enable
2404c
POPCNT CIRCUIT
908c
RESULT
OUT 2407c
b47
b32
Source1IN
2406d
Enable
2404d
POPCNT CIRCUIT
2408d
RESULT
OUT 2407d
b63
b48
Result
BUS 2409

Figure 24

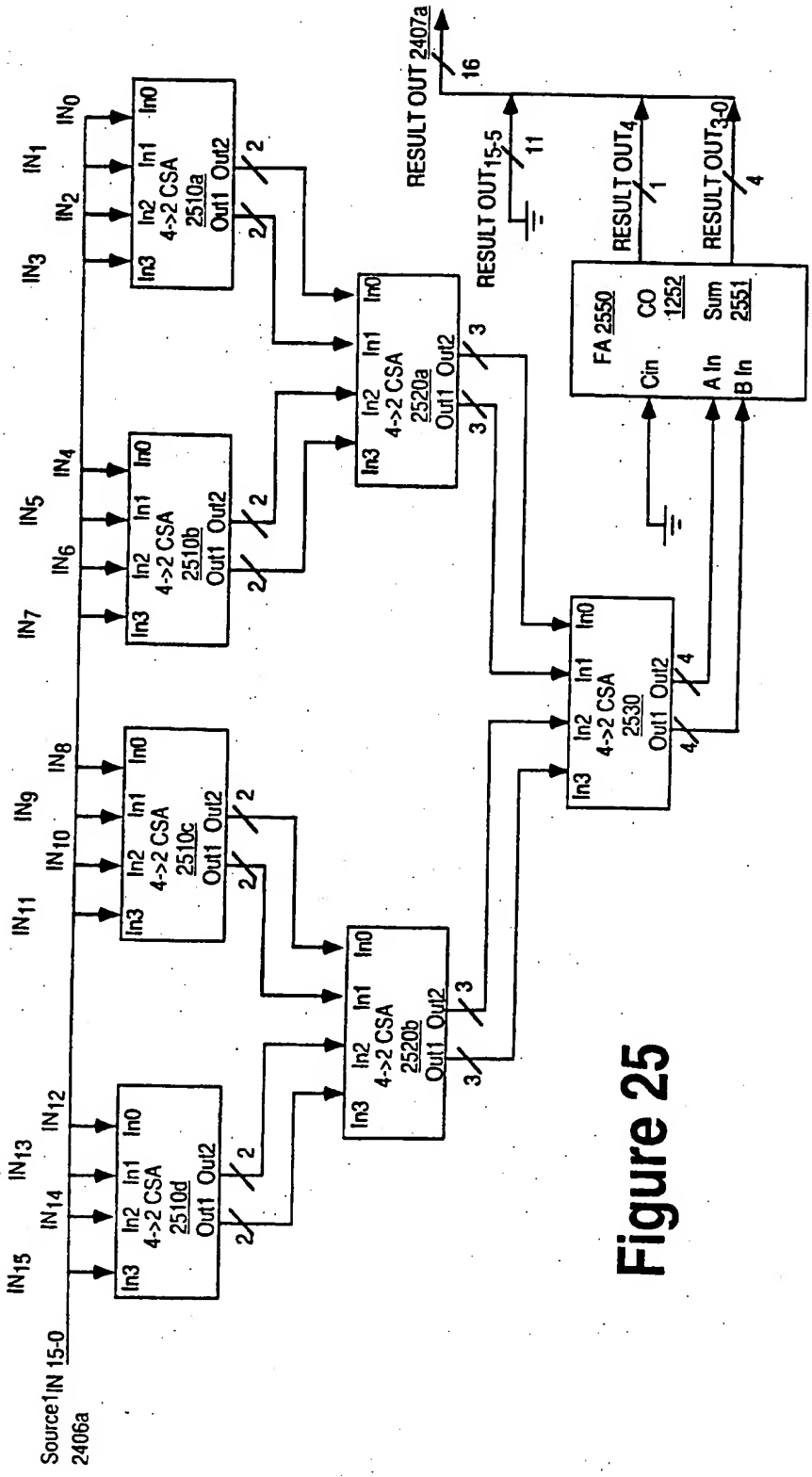

Source1 IN 15-0
2406a
IN15
IN14
IN13
IN12
IN11
IN10
IN9
IN8
IN7
IN6
IN5
IN4
IN3
IN2
IN1
IN0
In3 In2 In1 In0
4->2 CSA
2510d
Out1 Out2
2510c
2510b
2510a
2520b
2520a
2530
RESULT OUT 2407a
16
RESULT OUT15-5
11
RESULT OUT4
1
RESULT OUT3-0
4
FA 2550
Cin
CO 1252
A In
Sum 2551
B In

Figure 25

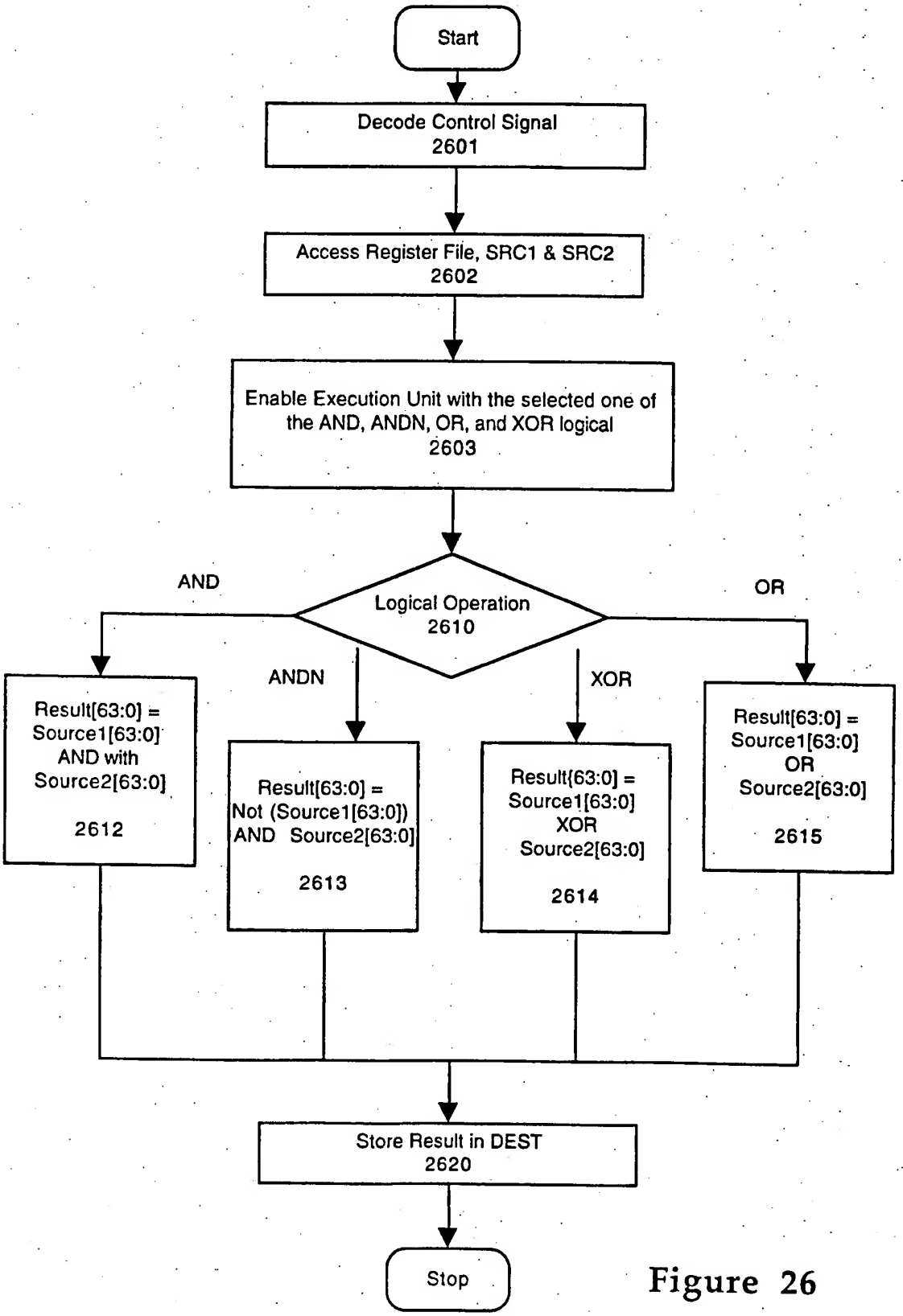

Start
Decode Control Signal
2601
Access Register File, SRC1 & SRC2
2602
Enable Execution Unit with the selected one of the AND, ANDN, OR, and XOR logical
2603
Logical Operation
2610
AND
ANDN
XOR
OR
Result[63:0] = Source1[63:0] AND with Source2[63:0]
2612
Result[63:0] = Not (Source1[63:0]) AND Source2[63:0]
2613
Result[63:0] = Source1[63:0] XOR Source2[63:0]
2614
Result[63:0] = Source1[63:0] OR Source2[63:0]
2615
Store Result in DEST
2620
Stop

Figure 26

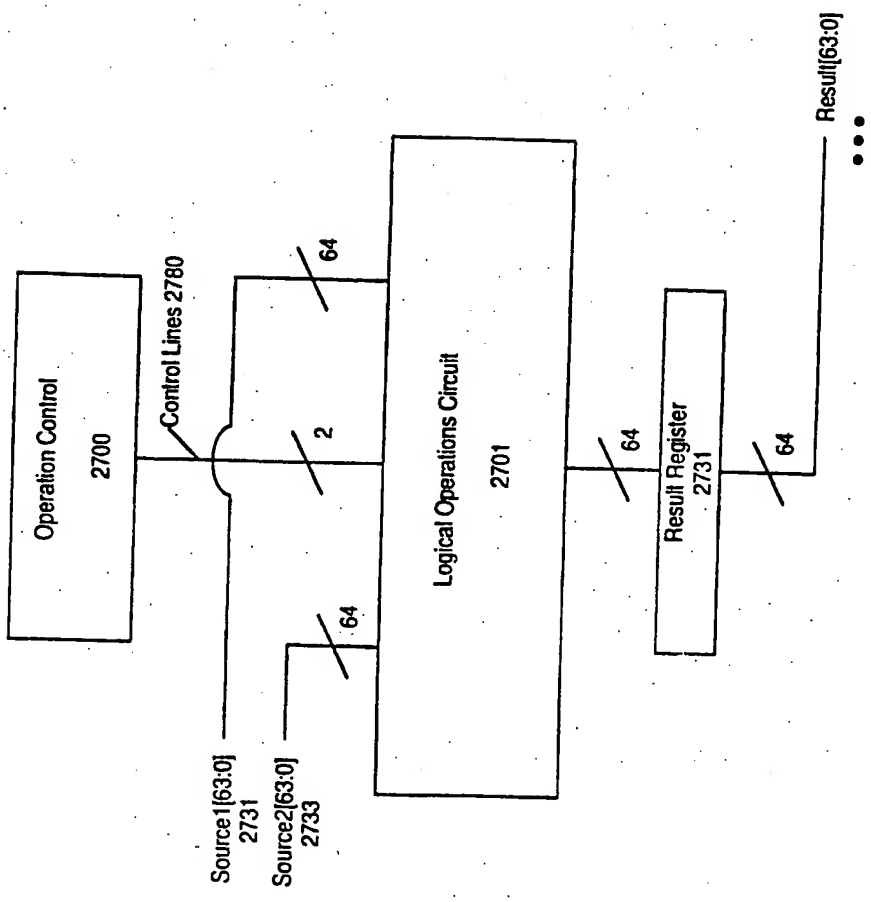

Operation Control
2700
Control Lines 2780
64
2
64
Logical Operations Circuit
2701
64
Result Register
2731
64
Result[63:0]
Source1[63:0]
2731
Source2[63:0]
2733

Figure 27

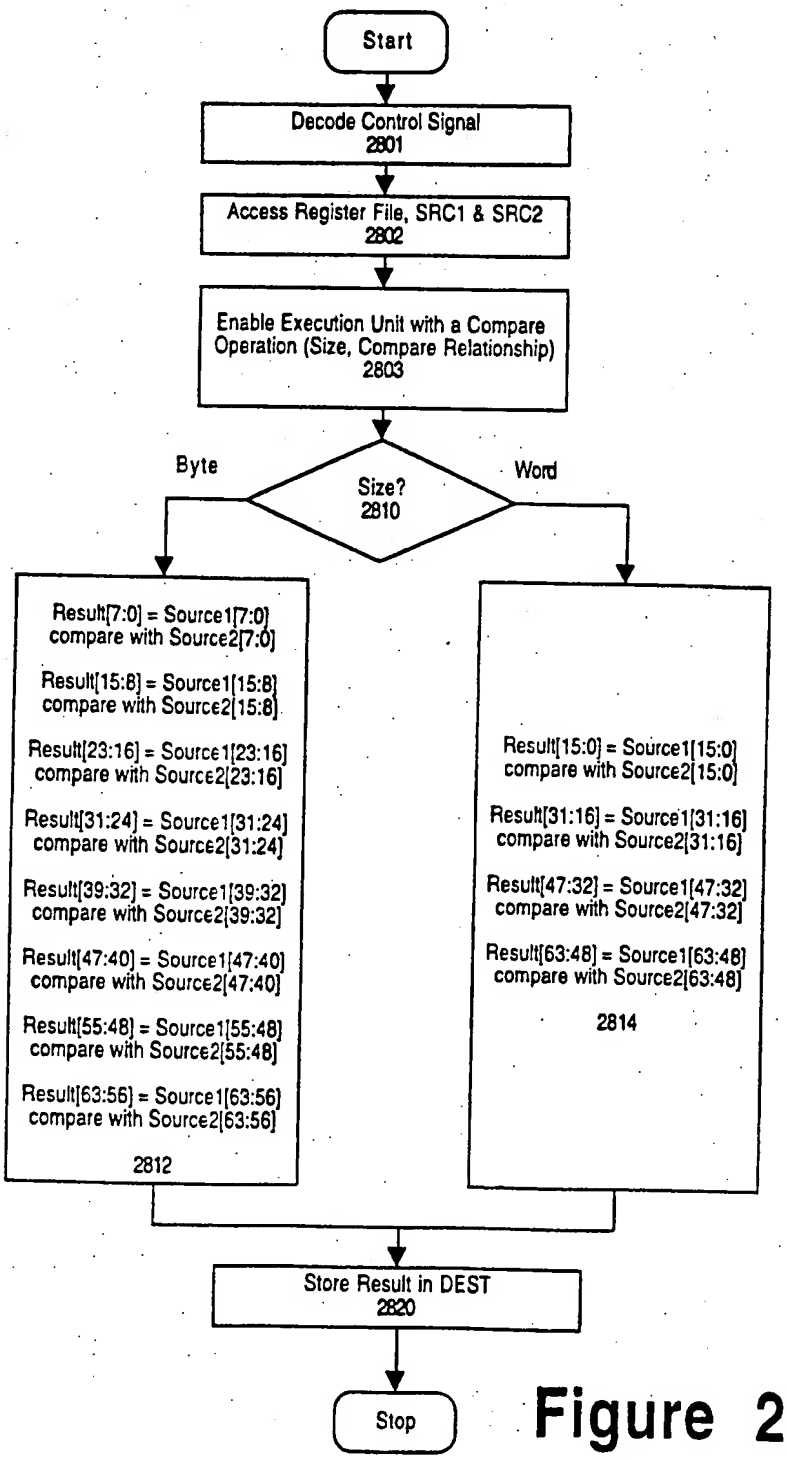
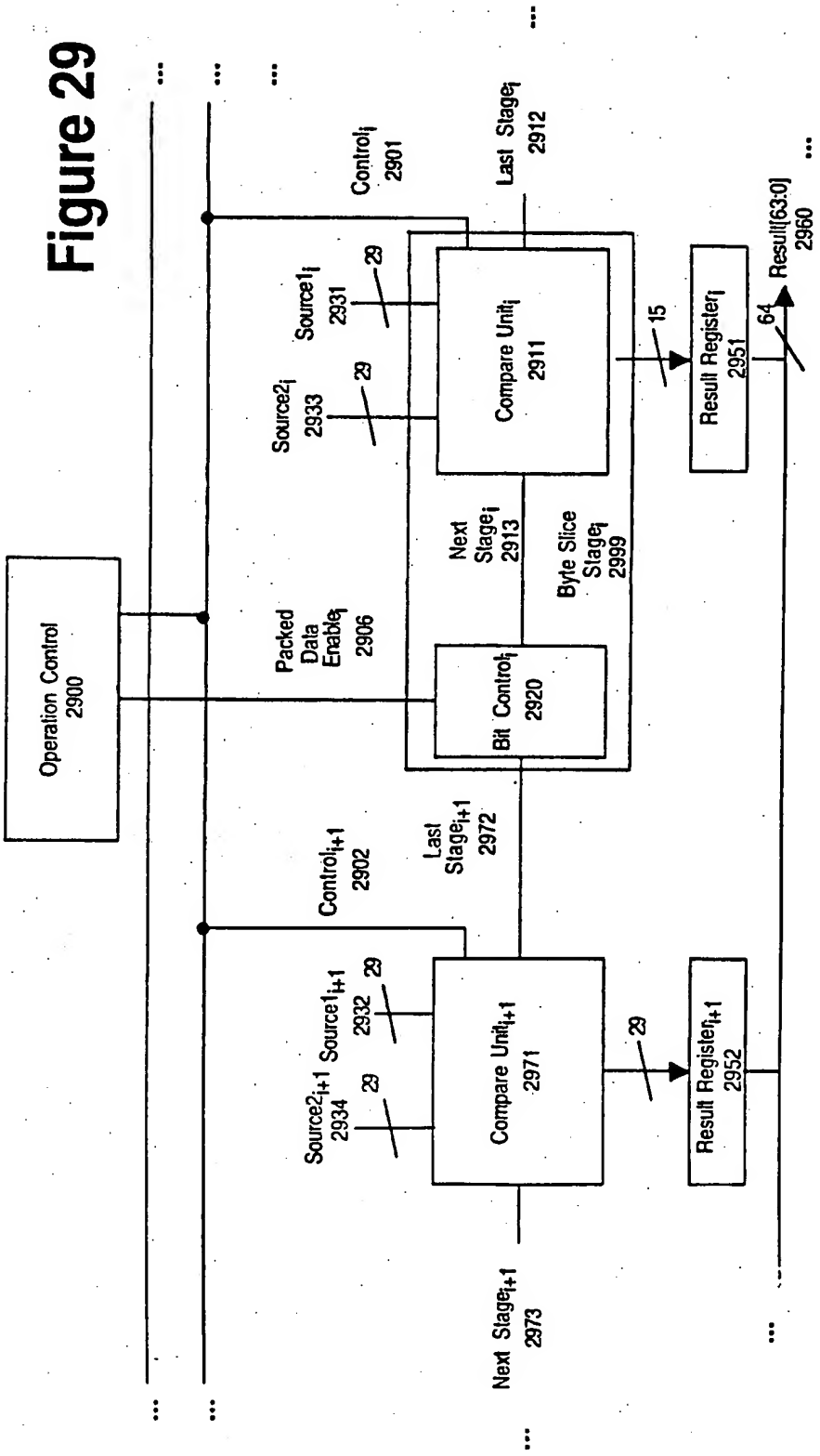

Start
Decode Control Signal
2801
Access Register File, SRC1 & SRC2
2802
Enable Execution Unit with a Compare Operation (Size, Compare Relationship)
2803
Size?
2810
Byte
Word
Result[7:0] = Source1[7:0] compare with Source2[7:0]
Result[15:8] = Source1[15:8] compare with Source2[15:8]
Result[23:16] = Source1[23:16] compare with Source2[23:16]
Result[31:24] = Source1[31:24] compare with Source2[31:24]
Result[39:32] = Source1[39:32] compare with Source2[39:32]
Result[47:40] = Source1[47:40] compare with Source2[47:40]
Result[55:48] = Source1[55:48] compare with Source2[55:48]
Result[63:56] = Source1[63:56] compare with Source2[63:56]
2812
Result[15:0] = Source1[15:0] compare with Source2[15:0]
Result[31:16] = Source1[31:16] compare with Source2[31:16]
Result[47:32] = Source1[47:32] compare with Source2[47:32]
Result[63:48] = Source1[63:48] compare with Source2[63:48]
2814
Store Result in DEST
2820
Stop

Figure 28

Figure 29


Operation Control 2900
Control$_i$ 2901
Source1$_i$ 2931
Source2$_i$ 2933
Packed Data Enable$_i$ 2906
Compare Unit$_i$ 2911
Last Stage$_i$ 2912
Next Stage$_i$ 2913
Byte Slice Stage$_i$ 2999
Bit Control$_i$ 2920
Result Register$_i$ 2951
Result[63:0] 2960
Control$_{i+1}$ 2902
Last Stage$_{i+1}$ 2972
Source1$_{i+1}$ 2932
Source2$_{i+1}$ 2934
Compare Unit$_{i+1}$ 2971
Result Register$_{i+1}$ 2952
Next Stage$_{i+1}$ 2973
29
15
64

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/11893

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 7/00, 7/38, 7/52, 7/50, 9/30
US CL :364/715.08, 736, 757, 769, 784; 395/375
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 364/715.08, 736, 757, 769, 784, 725, 726, 749, 754, 758, 760; 395/375

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS
Search terms: packed, data, pack, unpack, add, compare, shift, multiply

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	MC88110UM/AD Second Generation-RISC Microprocessor User's Manual, September 1992, pages 1-1 through 1-23, 5-1 through 5-25 and 10-62 through 10-71.	1-24
Y	US, A, 4,985,848 (PFEIFFER ET AL) 15 January 1991, Cols. 38-57 and Figs. 15-29.	1-24
A	US, A, 4,811,269 (HIROSE ET AL) 07 March 1989, Abstract	1-24
A	US, A, 5,126,964 (ZURAWSKI) 30 June 1992, Abstract.	1-24
A	US, A, 5,001,662 (BAUM) 19 March 1991, Abstract.	1-24

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 04 SEPTEMBER 1996	Date of mailing of the international search report 27 SEP 1996
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer CHUONG D. NGO Telephone No. (703) 305-3800

Form PCT/ISA/210 (second sheet)(July 1992)*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/11893

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US, A, 4,760,545, (INAGAMI ET AL) 26 July 1988, Abstract.	1-24